National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

NL-339 (r. 88/04)

Canada

THE UNIVERSITY OF ALBERTA

FINITE ELEMENT SOLUTION OF POLLUTANT CONSERVATION EQUATION AND ITS APPLICATION TO RIVER PLUMES

BY

SANDEEP C. SOLANKI

(C)

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF CIVIL ENGINEERING

EDMONTON, ALBERTA

FALL, 1988

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-45643-4

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: SANDEEP C. SOLANKI

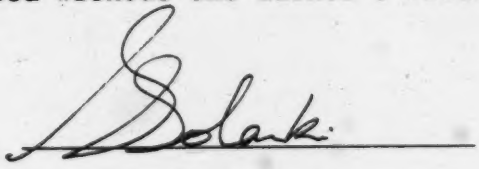
TITILE OF THESIS: FINITE ELEMENT SOLUTION OF THE POLLUTANT CONSERVATION EQUATION AND ITS APPLICATION TO RIVER PLUMES

DEGREE: MASTER OF SCIENCE

YEAR THIS DEGREE GRANTED: FALL 1988

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis or extensive extracts from it may be printed or otherwise reproduced without the author's written permission.



Permanent Address:

827, Woodpark Way S.W.
Calgary, Alberta
Canada, T2W 2T9

Date: June 15, 1988

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled FINITE ELEMENT SOLUTION OF POLLUTANT CONSERVATION EQUATION AND ITS APPLICATION TO RIVER PLUMES submitted by SANDEEP C. SOLANKI in partial fulfilment of the requirements for the degree of MASTER OF SCIENCE in CIVIL ENGINEERING.

SUPERVISOR

DATE: June, 2nd, 1988

ABSTRACT

A finite element formulation is used to solve the depth averaged pollutant conservation equation for a river. Two newly developed higher order upwinding elements are tested against standard methods. These new elements performed well in one and two dimensional purely advection tests. Their performance was a great deal better at higher Peclet numbers, in their ability to predict flows skewed to the mesh. The application of these new elements to both ideal and practical situations is restricted due to the large demands they make on computation effort.

The model was used to better understand the variation of eddy diffusivity on the characteristics of plumes. The performance of the model was adequate when it was applied to the solution of a plume in the Grand River, near Kitchener, Ontario. The application to the slug test was not as good, as the physics of the processes involved are not yet understood.

In conclusion the finite element application to river plume was found to be an excellent tool due to its flexibility in incorporating arbitrary geometries, and its higher accuracy in a general case.

ACKNOWLEDGEMENTS

The author wishes to acknowledge the encouragement and support provided by Dr. N. Rajaratnam.

The assistance provided by Dr. P. Steffler in the development and implementation of the programming and numerical modelling aspects of the thesis is greatly appreciated.

The encouragement given by Dr. P Steffler and Mr. S. Lovell in use of the latest computation hardware available to perform many aspects of this thesis is also acknowledged.

This thesis was financially supported by the Natural Sciences and Engineering Research Council by means of grants to Dr. N. Rajaratnam, and a scholarship from the Government of Alberta.

Table of Contents

Chapter **Page**

1. INTRODUCTION 1

2. LITERATURE REVIEW 2

2.1 Depth averaged Pollutant Conservation Equation ... 2

2.1.1 Pollutant Conservation Equation 2

2.1.2 Depth Averaged Equations 5

2.1.3 Diffusion Processes 5

2.1.3.1 Transverse Diffusion 6

2.1.3.2 Longitudinal Diffusion 10

2.1.4 Dispersion Processes 10

2.1.4.1 Final 'Steady' Dispersion 12

2.1.4.2 Initial Transient Dispersion 13

2.1.5 Modelled Equation 14

2.2 Past Solution Techniques 14

2.2.1 Analytical Solutions 15

2.2.1.1 One Dimensional Solutions 15

2.2.1.2 Two Dimensional Solutions 16

2.2.2 Numerical Solutions 20

2.2.2.1 Artificial Diffusion 21

2.2.2.1.1 von Neumann Stability Analysis 22

2.2.2.1.2 Taylor Series Analysis 28

2.2.2.2 Popular Finite Difference Schemes 31

2.2.2.2.1 Stone and Brian Method 32

2.2.2.2.2 The Holly-Preissmann Method 37
2.2.2.3 Finite Element Methods 39
3. FINITE ELEMENT METHOD 40
3.1 Method of Weighted Residuals 41
3.1.1 The Weak Statement 41
3.1.2 Semi-Discrete Form 44
3.1.3 Coordinate Transformation 45
3.1.4 Time Discretization 48
3.1.5 Boundary Conditions 48
3.1.6 Discrete Equation 49
3.1.7 The Solution Algorithm 50
3.2 Choice of Test and Basis Functions 51
3.2.1 Choice of Basis and Test Functions 53
3.2.2 The Program 54
4. NUMERICAL TESTS 64
4.1 One Dimensional Comparison 64
4.1.1 LBG 64
4.1.2 QUPG 65
4.1.3 CUPG 71
4.2 Two Dimensional Tests 76
4.2.1 Two Dimensional Advection 77
4.2.2 Steady Plume in Skewed Meshes 83
4.2.2.1 Case 1 85
4.2.2.2 Case 2 88
4.3 Conclusion of Numerical Tests 91
5. APPLICATIONS 93

5.1 Variation of Eddy Diffusivity. 93

5.1.1 Center Discharges 94

5.1.2 Side Discharges 99

5.2 Application to River Plume Discharges 107

5.2 Application to River Slug Tests 121

6. CONCLUSIONS 127

REFERENCES 129

APPENDIX 1 133

LIST OF TABLES

Table **Page**

4.1 Accuracy Measurements for the Advection of a Gauss Hill 80

4.2 Summary of Skew test, Case 1 and Case 2 89

5.1 Hydraulic Data for the Grand River 111

5.2 Hydraulic Data for the Athabasca River below Fort McMurray 124

LIST OF FIGURES

Figure		Page
2.1	Definition Sketch with Cartisian Coordinate system	3
2.2	Relationship between Transverse diffusivity and aspect ratio	8
2.3	Relationship for Transverse diffusivity	9
2.4	Effects of a Logarithmic velocity distribution on the depth averaged concentration	11
2.5	Curvilinear Coordinate system	18
2.6	Dissipation and Dispersion for Central Difference	25
2.7	Advection of a pulse using Central Difference	27
2.8	Dissipation and Dispersion for Upwinding Finite Difference	29
2.9	Advection of a pulse using Upwinding Finite Difference	30
2.10	Dissipation and Dispersion for Stone and Brian	35
2.11	Advection of a pulse using Stone and Brian	36
2.12	Advection of a pulse using Holly and Preissmann	38
3.1	Coordinate Transformation Using Mapping Functions	47
3.2	Local Domain of the Linear Rectangular Element	56
3.3	Local Domain of the Quadratic Upwinding Rectangular Element	56
3.4	Local Domain of the Cubic Upwinding Rectangular Element	57
3.5	Flow Chart 1a, A General Flow Chart of the Program CDFEM	59

3.6 Flow Chart 1b, Generate a Mesh Fitted to a River Reach 60
3.7 Flow Chart 1c, The Programmed Solution Algorithm 61
3.7 Flow Chart 2a, Assembling of Global Matrices, Part I 62
3.8 Flow Chart 2b, Assembling of Global Matrices, Part II 63
4.1 Dissipation and Dispersion for LBG 66
4.2 Advection of a pulse using LBG 67
4.3 Dissipation and Dispersion for QUPG 69
4.4 Advection of a pulse using QUPG 70
4.5 Dissipation and Dispersion for CUPG 72
4.6 Advection of a pulse using CUPG 74
4.7 Comparison of 1-D tests 75
4.8 Circular Advection of a Gauss Hill at a Courant # ~ 1.9 78
4.9 Circular Advection of a Gauss Hill at a Courant # ~ 0.19 79
4.10 Error measures for 2-D advection Courant # ~ 1.9 82
4.11 Error measures for 2-D advection Courant # ~ 0.19 84
4.12 Solutions for different Skew angle Using LBG 86
4.13 Solutions for different Methods for 60° skew angle, Pe = 6330 89
4.14 Concentration Profile across the Solutions for Skew test 90
4.15 Computational effort required for the three methods 92

5.1 Variation of Exx, Ezz and U 96
5.2 Results for Triangular Channel 97
5.3 Plume Spreading Rates in a Triangular Cross-Section Channel 98
5.4 Concentration Profile in the Center of Plume Along the Channel 100
5.5 Mesh and Velocity Vectors 102
5.6 Contour Plots for Trapezoidal Shape Channel 103
5.7 Close-up of the Contours 104
5.8 Variation of bz, the plume half width along the channel 105
5.9 Concentration Profile along the Channel 106
5.10 Longitudinal Concentration Profiles at the Bank of the Channel for Cases b,c and d 108
5.11 A Sketch of Grand River near Kitchener, Ontario, Canada 109
5.12 Mesh, Velocity and Contours for Grand River 113
5.13 Concentration Profiles at different Stations Grand River near Kitchener Ontario Canada 114
5.14 Concentration Profile at L = 213m, Grand River 115
5.15 Concentration Profile at L = 463m, Grand River 116
5.16 Concentration Profile at L = 713m, Grand River 117
5.17 Concentration Profile at L = 963m, Grand River 118
5.18 Concentration Profile at L = 1213m, Grand River 119
5.19 Concentration Profile at L = 1463m, Grand River 120
5.20 Mesh, Velocity Vectors and Initial Condition for the Slug Test 124

5.21 Concentration Contours for a Slug test 125

5.22 Computed Concentration at x = 6360m, Athabasca River 126

LIST OF SELECTED SYMBOLS

$[J]$	- coordinate transformation matrix
$[K]$	- stiffness matrix
$[M]$	- mass matrix
A and B	- coefficients used to fix the boundary condition
C	- depth averaged concentration
c	- instantaneous concentration
$\overline{C}$	- cross-sectional averaged concentration
$\overline{c}$	- time averaged concentration
C_i	- nodal concentration
CUPG	- Cubic Upwinding Petrov Galerkin Method
D_{ij}	- the tensor of the molecular diffusion
$\overline{D_{ij}}$	- time averaged tensor of the molecular diffusion
ε_i	- eddy diffusivity
E_{ij}	- sum of dispersion and eddy diffusivity
E_x	- eddy diffusivity & dispersion in the x direction
E_z	- eddy diffusivity & dispersion in the z direction
ϕ	- relative numerical dispersion
f_i	- basis functions
Γ	- the surface of the domain
g	- acceleration due to gravity
γ	- coefficients of the Fourier series Solution
h	- mean depth
j	- complex number $\sqrt{-1}$
k	- roughness
K_{ij}	- dispersion tensor

λ - wavelength of the perturbations

L2 - L-2 norm

L2' - L-2 discrete norm

LBG - Linear Bubnov Galerkin Method

QUPG - Quadratic Upwinding Petrov Galerkin Method

L_i - length to where the slug occupies entire channel

m_i - metric coefficients to account for a curvilinear coordinate system

p' - depth averaged source or sink term

p - source or sink term

$\bar{p}$ - time averaged source or sink term

P_e - Peclet number

θ - implicit coefficient

q_c - cumulative discharge

R - hydraulic radius

ρ - Courant number

r - amplification component of the Fourier Mode

ρ_m - density

S_f - slope of the specific energy line

U - depth averaged velocity in x direction

$\overline{u'c'}$ - time averaged velocity and concentration fluctuations

$\overline{\overline{u'c'}}$ - depth and time averaged velocity in the x direction and concentration fluctuations

U_* - shear velocity

$\overline{U}$ - cross-sectional averaged velocity in x direction

$\overline{u_D c_D}$ - deviations from the cross-sectional averaged velocity and concentration

$\overline{u_d c_d}$ - deviations from the depth averaged x direction velocity and concentration

u_i - three coordinate velocities

$\overline{u_i}$ - time averaged coordinate velocities

V - depth averaged velocity in the z direction

$\overline{\overline{v'c'}}$ - depth and time averaged velocity in the z direction and concentration fluctuations

$\overline{v_d c_d}$ - deviations from the depth averaged z direction velocity and concentration

v_i - test functions

W - channel width

ω - wave speed

W' - distance from the closets bank to the center of the plume

X_i - nodal coordinate in the x direction

Z_i - nodal coordinate in the z direction

$\{F\}$ - force matrix

Ω - domain of the problem

1. INTRODUCTION

The urbanization of our society in the recent century has brought with it the immense problem of liquid waste disposal. A common form of disposal currently used is to dilute the waste in a large body of water, which would then with time decompose in the water. To design disposal systems on a river the engineer must understand the physical behavior of pollutants in rivers. This understanding requires the development of efficient and accurate methods of analysis. This thesis attempts to further the use of the finite element method as a vital and essential tool in the engineers ability to effectively predict the behavior of passive pollutants in rivers.

2. LITERATURE REVIEW

2.1 Depth averaged Pollutant Conservation Equation

This section presents the derivation of the depth averaged advection diffusion equation which describes the behavior of a passive pollutant in rivers. A brief introduction to the physical processes important in the modelling of plumes is also included.

2.1.1 Pollutant Conservation Equation

The behavior of a conservative passive pollutant in an incompressible laminar fluid flow can be shown to be described differentially by the expression,

$$\frac{\partial c}{\partial t} + u_i \frac{\partial c}{\partial x_i} = \frac{\partial}{\partial x_i} D_{ij}\left(\frac{\partial c}{\partial x_j}\right) + p \quad . \tag{2.1}$$

In this expression c is the concentration of the pollutant, u is the velocity of the fluid, D is the molecular diffusion coefficient and p represents a source or a sink of pollutant. The index i represents the summation over the three Cartesian coordinate as shown in Figure 2.1.

This equation is only applicable to laminar flow, but it also represents the conservation of pollutants in turbulent flows at an instant in time and space. The time and space interval over which this equation is valid is so small that, until new faster supercomputers are available, solution of this equation for a turbulent flow situation is impossible.

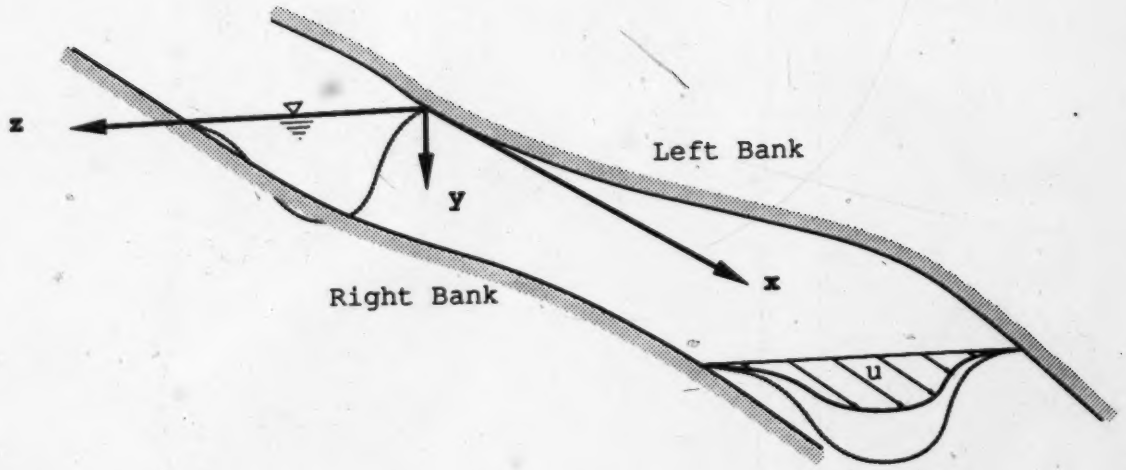


Figure 2.1 Definition Sketch with the Cartisian Coordinate System

In general this equation is time averaged to arrive at an equation applicable to turbulent flows. To time average Equation 2.1, the variable at an instant of time is expressed as a sum of its time averaged and fluctuating components, stated as $u_i = \bar{u}_i + u'$ and $c = \bar{c} + c'$. The time averaged Equation 2.2 is arrived at after substituting into Equation 2.1 these expressions and simplifying.

$$\frac{\partial \bar{c}}{\partial t} + \bar{u}_i \frac{\partial \bar{c}}{\partial x_i} = \frac{\partial}{\partial x_i}\left(- \overline{u_i'c'}\right) + \frac{\partial}{\partial x_i} \overline{D_{ij}} \left(\frac{\partial \bar{c}}{\partial x_i}\right) + \bar{p} \qquad (2.2)$$

In this equation the bars denote time averaged quantities defined as

$$\overline{(.)} = \frac{1}{T} \int_t^{t+T} (.) \; dt, \qquad (2.3)$$

and the primes indicate fluctuating components of the respective quantities. The additional term on the right hand side $\overline{u_i'c'}$, represents the turbulent flux of fluctuating concentrations carried by the fluctuating velocity field. These terms describe the dominant mixing processes which dilute the effluent in the river.

There exists no general solution for Equation 2.2 for situations that are of concern to the engineer. The engineer is forced to devise an approximate solution for problems he is concerned with.

2.1.2 Depth Averaged Equations

A practical alternative that has been used in the past is to solve the depth averaged equation and apply it to situations where the mixing with respect to the depth is fully accomplished (Beltaos 1978, Fisher *et al* 1979, Harden *et al* 1979, Krishnappan *et al* 1983 and Sayre 1973). The depth averaged equation can be derived by integrating Equation 2.2 with respect to the depth and dividing the resulting equation by the depth. The resulting depth averaged equation is

$$\frac{\partial C}{\partial t} + U\frac{\partial C}{\partial x} + V\frac{\partial C}{\partial z} = \frac{\partial}{\partial x}\left(-\overline{\overline{u'c'}} - \overline{u_d c_d}\right) + \frac{\partial}{\partial y}\left(-\overline{\overline{v'c'}} - \overline{v_d c_d}\right) + P \quad . \qquad (2.4)$$

In this equation C, U, V, and P are the depth averaged quantities defined as $\frac{1}{h}\int_0^h (.)\, dy$. The depth averaging dispersion terms $\overline{u_d c_d}$ and $\overline{v_d c_d}$ represent the effect of the non-uniform velocity distribution with respect to the depth. The quantities u_d, v_d and c_d are the deviations from the average for each quantity. The terms $-\overline{\overline{u'c'}}$ and $-\overline{\overline{v'c'}}$ are the depth and time averaged turbulent fluctuations.

2.1.3 Diffusion Processes

The time and depth averaging results in two new terms, $-\overline{\overline{u'c'}}$ and $-\overline{\overline{v'c'}}$, that represent the mixing caused by the turbulent fluctuations. In his pioneering paper, G. I.

Taylor (1921) showed that momentum turbulent fluctuations can be modelled as a Fickian behavior, given that the turbulent time scale is much smaller than the time scale of the general flow. Similarly the fluctuating components in the time averaged pollutant conservation equation can also be modelled assuming Fickian behavior. This Fickian process implies that the fluctuation components can be modelled by a simple gradient law expression such as shown below (Elhdi *et al* 1984),

$$-\rho_m \overline{\overline{u_i' c'}} = \varepsilon_{ij} \frac{\partial \bar{c}}{\partial x_j} \quad (2.5)$$

Where ε_i represents the eddy diffusivity in the three respective directions. From dimensional considerations it can be deduced that the eddy diffusivity is a product to a length scale and a velocity scale (Krishnappan and Lau 1977). The length scale used generally is either the depth or the width of the river and the velocity scale used is either the reach average velocity or the shear velocity $U_* \equiv \sqrt{gRS_f}$.

2.1.3.1 Transverse Diffusion

The most important factor governing the mixing process in a steady plume in a river is the transverse mixing of the plume in the fluid flow. This diffusive process governs the rate of lateral spreading of the plume.

Through the use of dimensional arguments a relationship that governs the behavior of ε_z, the transverse eddy

diffusivity, was looked at in great detail by Krishnappan and Lau (1977). They concluded that the best fit to the experimental values documented is given by the relationship

$$\frac{\varepsilon_z}{U_* W} = f\left(\frac{U}{U_*}, \frac{W}{h}\right) \qquad (2.6)$$

where W is the average width of the river and h is the reach average depth. Figure 2.2 shows a plot of this relationship as given by Krishnappan and Lau (1977). The original plot presented by Krishnappan and Lau shows that there is a strong dependence on friction factor. A convenient expression that accounts for this 'roughness' effect can be derived if the expression

$$\frac{U}{U_*} = 2.5 \ln\left(\frac{12R}{k}\right) \qquad (2.7)$$

is used to incorporate the relationship between the hydraulic radius R ($\cong$ h for a river)and the conveyance coefficient $C_* \equiv \frac{U}{U_*}$. In this equation, k is the roughness height of the channel bed. A possible dimensionless relationship resulting from this is

$$\frac{\varepsilon_z}{UW} = f\left(\frac{k}{W}\right) \qquad (2.8)$$

The experimental data is plotted again using this new relationship in Figure 2.3 . This plot suggests that an approximate value can be obtained from the linear regression line,

$$\frac{\varepsilon_z}{UW} = 0.0006538 + 0.0195\left(\frac{k}{W}\right). \qquad (2.9)$$

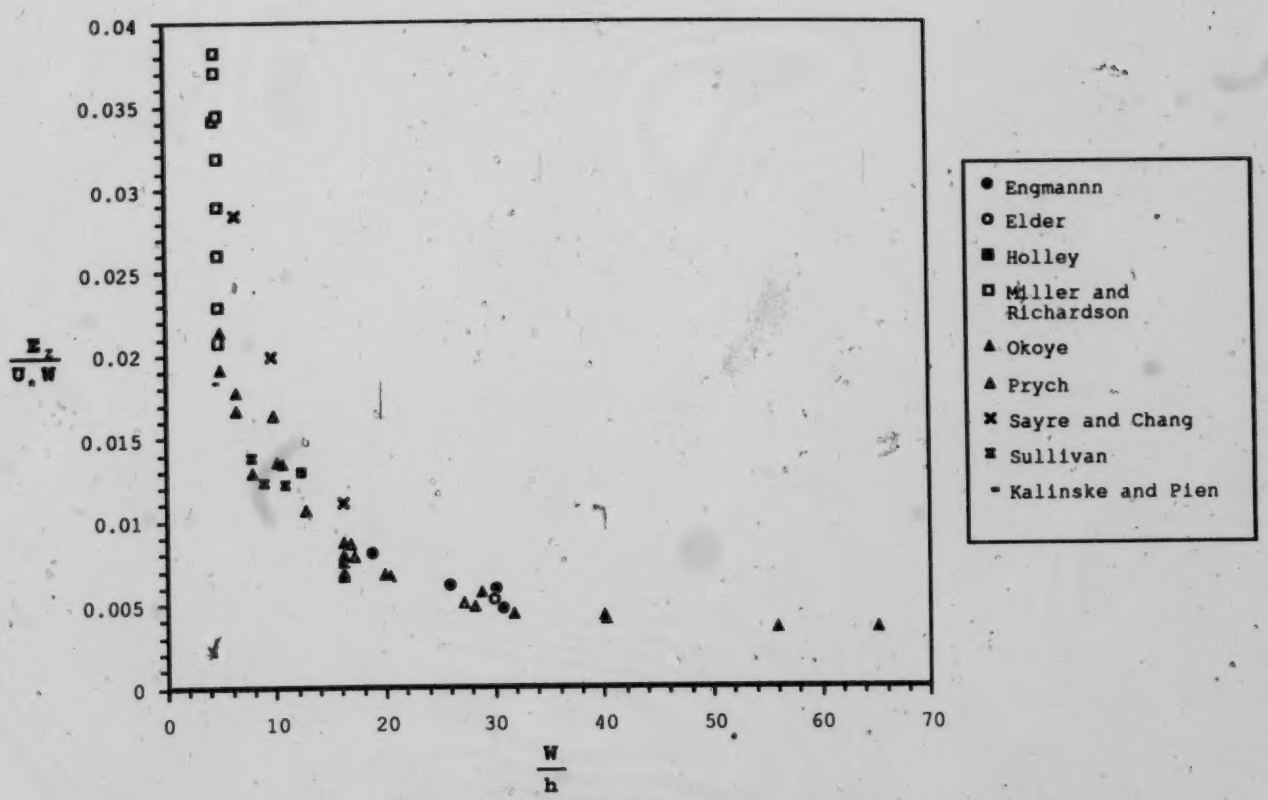


Figure 2.2 Relationship between transverse diffusivity and aspect ratio (modified from Krishnappan and Lau (1977))

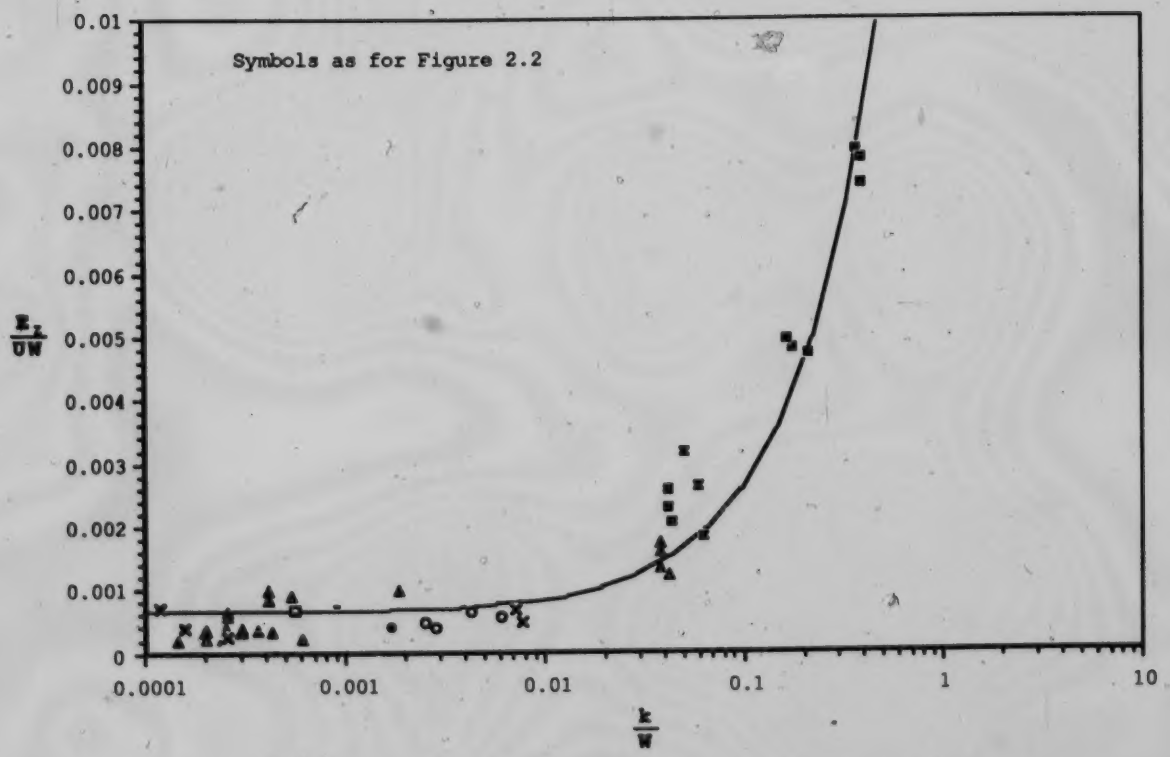


Figure 2.3 Relationship for Transverse Diffusivity

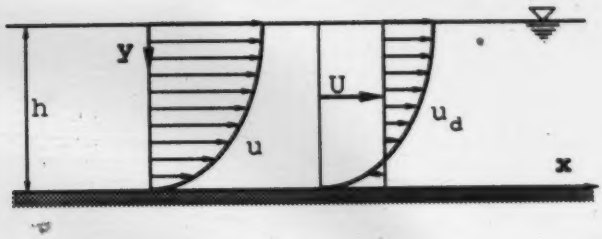
2.1.3.2 Longitudinal Diffusion

Longitudinal diffusivity can also be thought to follow a similar relationship as the transverse diffusivity. Even though this has not been studied in as extensively as the transverse diffusion, studies done by Sayre and Chang (1968) suggest that it is two to three times larger than the transverse diffusivity. The longitudinal diffusivity can be approximated in a similar manner by the expression

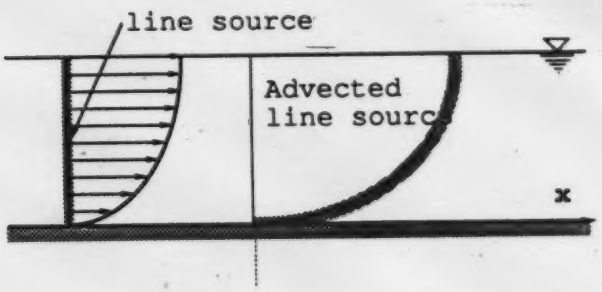
$$\frac{\varepsilon_x}{UW} = 0.0006538 + 0.0293 \left(\frac{k}{W} \right) . \qquad (2.10)$$

2.1.4 Dispersion Processes

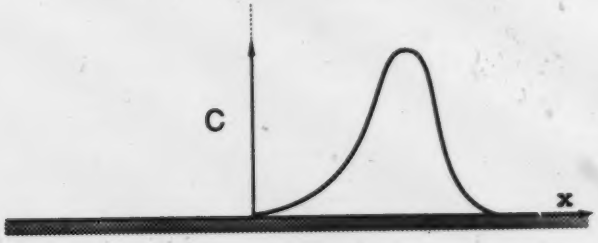
The depth averaged concentration equation hides the effects of any variation of properties in the vertical direction. One important distortion resulting from depth averaging is the smearing of pollutant in plan by the non-uniform vertical velocity distribution. Take the case of the release of a trace of pollutant in an infinitely wide channel, which has a logarithmic velocity distribution in the vertical. The tracer will be advected downstream at different velocities depending on its the distance above the bed. The result is that the cloud disperses in the horizontal plane, as illustrated in Figure 2.4. This dispersion manifests itself in the depth averaged concentration measurements, by showing a distinct skewness to the depth averaged concentration distribution. After a long



(a) Velocity Deviations for a logarithmic distribution.



(b) Advection of a line source by the logarithmic velocity distribution.



(c) Skewed distribution of the depth averaged concentration resulting from a non-uniform velocity distribution.

Figure 2.4 Effects of a logarithmic velocity distribution on the Depth averaged concentrations.

period of time this skewed distribution deteriorates into a Gaussian distribution. The effects of this behavior is incorporated by modelling the two dispersion terms , $\overline{u_d c_d}$ and $\overline{v_d c_d}$, which result from the depth averaging.

There exist two distinct regions of interest in the modelling of these dispersion terms. They consist of the initial developing region and a final 'steady' region.

2.1.4.1 Final 'Steady' Dispersion

In the final 'steady' stage of dispersion of the pollutant, it is believed that the variance of distribution grows linearly with time (Fisher *et al.* 1979). G. I. Taylor (1921) showed that if the variance can be thought of as growing linearly with time, that is

$$\frac{d\sigma}{dt} = \text{constant}, \qquad (2.11)$$

then it follows that the process can be modelled as a Fickian behavior. Thus the dispersion terms can be modelled as,

$$\overline{u_d c_d} = - K_{xx} \frac{dC}{dx} - K_{xz} \frac{dC}{dz} \text{ and}$$

$$\overline{v_d c_d} = - K_{zx} \frac{dC}{dx} - K_{zz} \frac{dC}{dz}, \qquad (2.12)$$

where K is defined by the following integrals,

$$K_{xx} = - \int u_d \int \frac{1}{\varepsilon} \int u_d \, dy \, dy \, dy,$$

$$K_{xz} = - \int u_d \int \frac{1}{\varepsilon} \int v_d \, dy \, dy \, dy,$$

$$K_{zx} = - \int v_d \int \frac{1}{\varepsilon} \int u_d \, dy \, dy \, dy \text{ and}$$

$$K_{zz} = - \int v_d \int \frac{1}{\varepsilon} \int v_d \, dy \, dy \, dy, \qquad (2.13)$$

In these integrals ε is the vertical turbulent eddy diffusion. Elder (1959) showed that for a logarithmic velocity distribution in the vertical direction in an infinitely wide channel, the corresponding expression for the dispersion coefficient in the x direction is

$$K_{xx} = 5.93 \, U_* \, h \qquad (2.14)$$

where U_* is the shear velocity and h is the depth.

This final region is defined to be after the cloud has travelled a characteristic length, after which the only dominant process is one dimensional dispersion. For a river this occurs after the slug occupies the full cross section. Fisher(1979) suggests that this occurs after a length,

$$L_i = \frac{1.8 \, W'^2 \, U}{h \, U_*}, \qquad (2.15)$$

where W' is the distance to the farthest bank from the center of the slug and U is the cross-sectional average velocity. Prior to this initial region, the use of a Fickian approximation is invalid.

2.1.4.2 Initial Transient Dispersion

In the initial region of the slug movement, there has not been any physically sound analysis performed. An attempt was made by S. Beltaos(1980) to define this region using an empirical fit to predict the behavior of the slug variance

for the initial region. Though his method of analyses is quite firm, the use of an empirical relation that has no physical basis makes the analyses unattractive.

The lack of a sound predictive model for the physical behavior of dispersion limits the ability of numerical methods in predicting the measured values for river slug tests.

2.1.5 Modelled Equation

The depth averaged equation modelled after the simplificatic.ı made for the dispersion and eddy diffusivity terms is

$$\frac{\partial C}{\partial t} + U_i \frac{\partial C}{\partial x_i} - \frac{\partial}{\partial x_i} E_{ij} \frac{\partial C}{\partial x_j} - P = 0 \qquad (2.16)$$

where E_{ij} is the total diffusion defined as

$$E_{ij} = \varepsilon_{ij} + K_{ij} \; . \qquad (2.17)$$

Equation 2.16 is applicable to situations where the plume is well mixed in the vertical direction and the longitudinal dispersion can be neglected.

2.2 Past Solution Techniques

The full three dimensional equation of conservation of a passive pollutant has no general solution for any situation. The solution techniques vary from the simple analytical solutions for idealized situations, to the more complicated

finite element discrete solution of a general river reach. The strategies that have been employed by the engineer to solve the problem can be divided into two basic categories, namely analytical solutions and numerical solutions of simplified forms of Equation 2.1.

2.2.1 Analytical Solutions

Analytical solution may not be as precise in reflecting the behavior in a general river as some of the other numerical solutions, but they do provide an excellent tool in assessing initial investigations. Basically the analytical solutions can be categorized into two groups, namely one dimensional and two dimensional solutions.

2.2.1.1 One Dimensional Solutions

Generally the one dimensional solutions are used to predict the behavior of a slug test. In a slug test a large dose of pollutant is introduced into the river. The dilution of the slug due to diffusion and dispersion can be predicted using a one dimensional solution. Once the slug has occupied the entire channel width the dominant processes that are of interest are those of advection and longitudinal dispersion of the slug as it travels downstream.

To solve for this situation, the full equation has to be averaged over the cross section to arrive at the unsteady one dimension advection diffusion equation,

$$\frac{\partial \bar{C}}{\partial t} + \bar{U} \frac{\partial \bar{C}}{\partial x_i} = \frac{\partial}{\partial x_i} \bar{E}_x \left(\frac{\partial \bar{C}}{\partial x_i} \right) \qquad (2.18)$$

In this equation $\bar{C}$, $\bar{U}$, and $\bar{E}_x$ are quantities averaged over the cross section. The averaged terms are defined as

$$\bar{C} = \int_0^W \int_0^h \bar{c} \, dy \, dz \qquad (2.19)$$

and $\bar{E}_x$ is the longitudinal dispersion defined as

$$\bar{E}_x = \frac{-\rho \overline{u_D \, c_D}}{\frac{\partial \bar{C}}{\partial x}} \qquad (2.20)$$

where u_D and c_D are the deviations from the cross-sectional average values. This dispersion due to the velocity profiles can only be modelled as a diffusive process using gradient laws if they behave in a Fickian manner. As mentioned earlier in Section 2.1.4.1, the distance after which this analysis is applicable is L_i.

The analytical solution to Equation 2.18 is

$$\bar{C} = \frac{M}{2 A \sqrt{\pi \bar{E}_x t}} e^{\left(\frac{-(x-\bar{U} t)}{4 \bar{E}_x t} \right)} \qquad (2.21)$$

where M is the total quantity of tracer introduced into the river, and A is the average cross-sectional area.

2.2.1.2 Two Dimensional Solutions

Analytical solutions to problems based on solving the depth averaged equation similar to Equation 2.4 exist for

steady plumes from line and point sources. Equation 2.4 is for a Cartesian coordinate system. However, no natural river is ever going to follow such an orthogonal coordinate system. To overcome this problem, Yotsukura and Sayre (1976) introduced a coordinate system fitted to the general shape of rivers, known as a body fitted curvilinear coordinate system, shown in Figure 2.5. The depth averaged equation in this coordinate system according to Yotsukura and Sayre can be expressed as,

$$m_x m_z \frac{\partial C}{\partial t} + \frac{\partial}{\partial x}(m_z UC) + \frac{\partial}{\partial z}(m_x VC) = \frac{\partial}{\partial x}\left(\frac{m_z}{m_x} E_x \frac{\partial C}{\partial x}\right) + \frac{\partial}{\partial z}\left(\frac{m_x}{m_z} E_z \frac{\partial C}{\partial z}\right) \qquad (2.22)$$

in which E_x and E_z are the sum of the turbulent diffusivity and dispersion coefficients, m_x and m_z are metric coefficients to account for the coordinate system transformation and all capitalized variables are depth averaged quantities.

The solution to Equation 2.22 may be obtained, but it would be of little benefit due to the large data base required. River velocity surveys seldom give V velocity measurements and for the purpose of preliminary analysis it has little effect. The solution generally used is a solution to a steady state situation where the longitudinal dispersion has little effect, since it is dependent on the longitudinal gradient which is very small. Applying these simplifications to Equation 2.22, the resulting equation can be stated as,

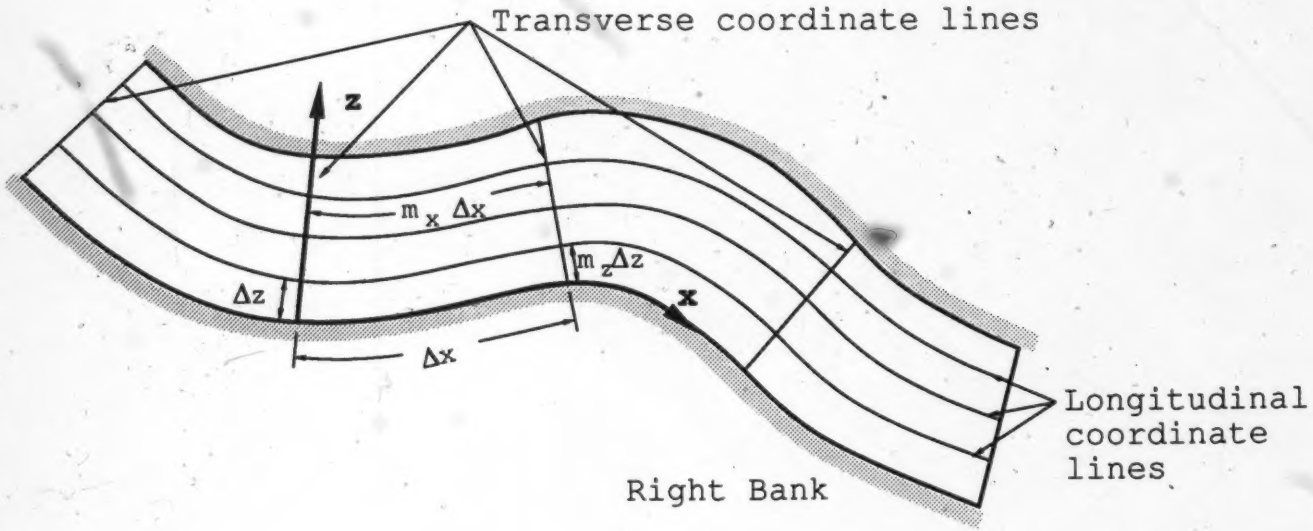


Figure 2.5 Curvilinear coordinate system

$$\frac{\partial}{\partial x}(m_z UC) = \frac{\partial}{\partial z}\left(\frac{m_x}{m_z} E_z \frac{\partial C}{\partial z}\right). \qquad (2.23)$$

Introducing the cumulative discharge concept defined as

$$q_c = \int_0^z m_z h U \, dz \qquad (2.24)$$

in place of z, Yotsukura and Sayre (1976) show that Equation 2.23 is the same as

$$\frac{\partial C}{\partial x} = \frac{\partial}{\partial q_c}\left(U h^2 m_z E_z \frac{\partial C}{\partial q_c}\right). \qquad (2.25)$$

The analytical solution of Equation 2.25 for a point source located at the bank, with constant E_z over the whole domain of the channel is

$$\frac{C}{C_\infty} = \frac{2}{\sqrt{2\pi\xi}}\left[e^{\left(\frac{-\eta^2}{2\xi}\right)} + \sum_{m=1}^{\infty}\left(e^{\left(\frac{-(2m-\eta)^2}{2\xi}\right)} + e^{\left(\frac{-(2m+\eta)^2}{2\xi}\right)}\right)\right] \qquad (2.26)$$

Where $C_\infty = \frac{M}{Q}$;

$\eta = \frac{q_c}{Q}$, the normalized cumulative discharge;

and $\xi = \frac{2Uh^2 m_z E_z x}{Q^2}$, a dimensionless distance.

A similar solution shown below exists for a line source extending from the bank to where the cumulative discharge, is equal to rQ,

$$\frac{C}{C_\infty} = \frac{1}{2r}\left[\operatorname{erf}\left(\frac{r-\eta}{\sqrt{2\xi}}\right) + \operatorname{erf}\left(\frac{r+\eta}{\sqrt{2\xi}}\right) + \sum_{n=1}^{\infty}\left\{\operatorname{erf}\left(\frac{2n+r-\eta}{\sqrt{2\xi}}\right) + \operatorname{erf}\left(\frac{2n+r+\eta}{\sqrt{2\xi}}\right)\right.\right.$$

$$\left.\left. - \operatorname{erf}\left(\frac{2n-r-\eta}{\sqrt{2\xi}}\right) - \operatorname{erf}\left(\frac{2n+r-\eta}{\sqrt{2\xi}}\right)\right\}\right] \qquad (2.27)$$

where the variables are defined as for the point source equation above.

Both of these solutions are only applicable to idealized situations where the transverse diffusivity is constant over the domain and the flow consists of U velocity only. The domain of the solution would also have an average velocity that is not radically different between various longitudinal distances.

2.2.2 Numerical Solutions

The analytical solutions discussed earlier have restrictions, in that the velocity and eddy diffusivity are averaged over the domain and the solutions cannot account for any irregularities in the flow field. To find solutions that give an accurate picture of the behavior in a natural channel, a numerical solution is necessary.

There are many numerical schemes, both finite difference and finite element, that have been applied to solve the general depth averaged Equation 2.13. Schemes that have been used in the past have been tested by trying to find an approximate solution as close as possible to the exact solution of the simple hyperbolic equation

$$\frac{\partial C}{\partial t} + U \frac{\partial C}{\partial x} = 0 \qquad (2.28)$$

The exact solution to this equation is simply

$$C(t_{new}, x) = C(t_{old}, x-x_t), \qquad (2.29)$$

where $x_\tau = \int_0^t U(\tau)\,d\tau$.

2.2.2.1 Artificial Diffusion

The problem of solving Equation 2.28 using any one of the many numerical methods available, is that to get a stable solution, the method generally introduces artificial diffusion. To understand why it is that a numerical solution should exhibit some artificial diffusion for a stable solution, we must first look at a solution algorithm that shows no artificial diffusion. One such algorithm can be derived by using a theta implicit finite difference approximation for the time derivative and a centered finite difference approximation for the spatial derivative. After some simplifications, the resulting discrete form of Equation 2.28 is,

$$C_i^{n+1} + \frac{\rho}{2}\theta\left[C_{i+1}^{n+1} - C_{i-1}^{n+1}\right] = C_i^n - \frac{\rho}{2}(1-\theta)\left[C_{i+1}^n - C_{i-1}^n\right] \qquad (2.30)$$

where ρ is the Courant number defined as $\frac{U_i^{n+1}\Delta t}{\Delta x}$, and θ is an implicit coefficient to determine where the space derivative is evaluated. $\theta = 0.5$ for a Crank Nicholson approximation, $\theta = 0.0$ for an explicit algorithm and $\theta = 1.0$ for a fully implicit algorithm.

2.2.2.1.1 von Neumann Stability Analysis

The stability of any algorithm like Equation 2.30 can be explored using the von Neumann stability analysis (Lapidus and Pinder, 1982). This analysis is restricted to discrete forms of the equation which are a linear representation at an ordinary point.

The analysis essentially consists of looking at one mode of the solution as expressed by a Fourier series approximation. The general Fourier series solution to Equation 2.28 can be expressed as

$$C(x,t) = \sum_{m=-\infty}^{\infty} \gamma_m e^{(i\,\omega\Delta x)} . \qquad (2.31)$$

Where γ_m are coefficients that are dependent upon the initial conditions imposed on Equation 2.28, i is the complex number $\sqrt{-1}$ and ω is the wave number defined as $\frac{2\pi}{\lambda}$ in which λ is the wavelength of a particular mode of the solution. The series solution represents a continuous solution, where as for a difference formula the solution is only solved for discrete points. The solution at any one discrete point(node) is,

$$C_i^n \equiv C(i\Delta x, n\Delta t) = \gamma^n e^{(i\omega(i\Delta x))} . \qquad (2.32)$$

Substituting expressions analogous to 2.32 into a discrete form and simplifying usually results in γ being a complex function of the $\omega\Delta x$. The real and imaginary parts of the complex function can be looked at by observing the

behavior of its magnitude and phase components. The magnitude r and relative phase ϕ are given by the general expressions

$$r = \sqrt{\gamma_{real}^2 + \gamma_{imaginary}^2} \quad \text{and} \tag{2.33}$$

$$\phi = \frac{\tan^{-1}\left(\frac{\gamma_{imaginary}}{\gamma_{real}}\right)}{\omega \Delta x \rho} . \tag{2.34}$$

To observe the behavior of these components, two plots are constructed, one of r verses N and another of ϕ verses N, where N is defined as $\frac{2\pi}{\omega \Delta x}$. Physically N defines the number of nodes used to represent an initial perturbation in the domain. The two plots physically represent the dissipation(r) and dispersion(ϕ) exhibited by the discrete representation of perturbation using N nodes.

A value of 1.0 for r and ϕ represent an ideal discrete algorithm which will propagate the perturbations at the exact speed(ϕ=1.0) with no modifications to their magnitude (r=1.0). Algorithms that exhibit $r > 1.0$ indicate an unstable solution since any perturbations would be amplified where as in the case for $r < 1.0$ they will damped out. For algorithms which exhibit values of $\phi < 1.0$ show dispersion where waves of high frequency(small wavelength) will be traveling at speeds lower than the exact.

Substituting appropriate expressions analogous to Equation 2.31 into the discrete Equation 2.30 and simplifying using the identity

$$e^{(-i\omega\Delta x)} = \cos(\omega\Delta x) - i \sin(\omega\Delta x) \qquad (2.35)$$

results in the following expression for γ

$$\gamma = \frac{\gamma_{real} + i\,\gamma_{imaginary}}{\gamma_{denominator}} \qquad (2.36)$$

where

$$\gamma_{real} = 1 - \theta(1-\theta)\left(\rho \sin(\omega\Delta x)\right)^2,$$

$$\gamma_{imaginary} = -\rho\left(\sin(\omega\Delta x)\right) \text{ and}$$

$$\gamma_{denominator} = 1 + \left(\theta\,\rho \sin(\omega\Delta x)\right)^2$$

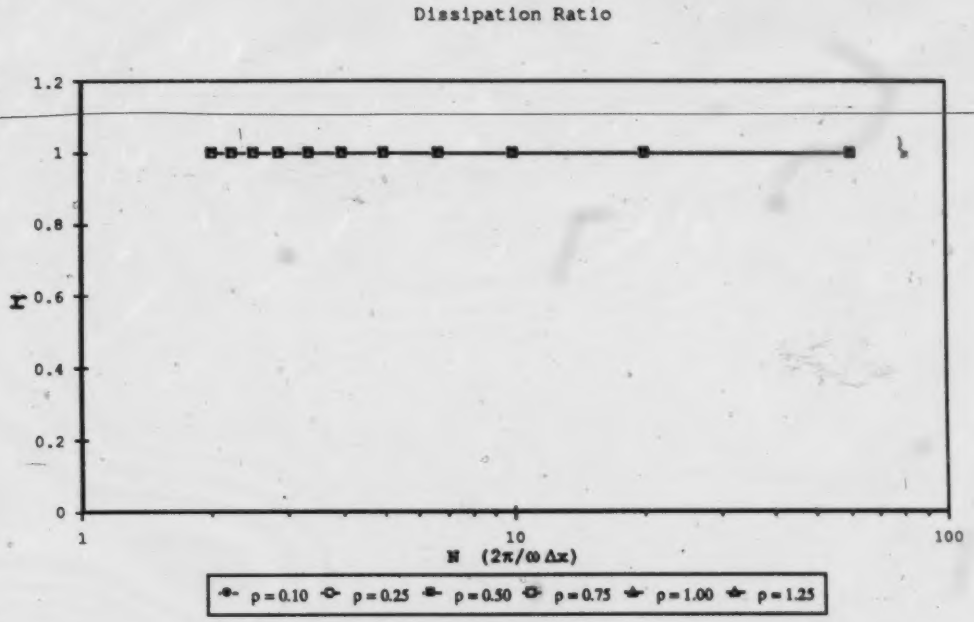
The corresponding expression for r and ϕ of the discrete form 2.30 can be interpreted as,

$$r = \sqrt{\left[\frac{\gamma_{real}}{\gamma_{denominator}}\right]^2 + \left[\frac{\gamma_{imaginary}}{\gamma_{denominator}}\right]^2} \qquad (2.37)$$

and

$$\phi = \frac{\tan^{-1}\left(\dfrac{\gamma_{imaginary}}{\gamma_{real}}\right)}{\omega\Delta x\rho}\,. \qquad (2.38)$$

Figure 2.6 shows the two plots of r and ϕ verses N for θ = 0.5 with Courant numbers varying from 0.1 to 1.25. These plots show that this particular formulation exhibits no dissipation, but does produce some dispersion. The inability of algorithms such as the central difference algorithm to restrict dispersion is the principal reason why they fail.



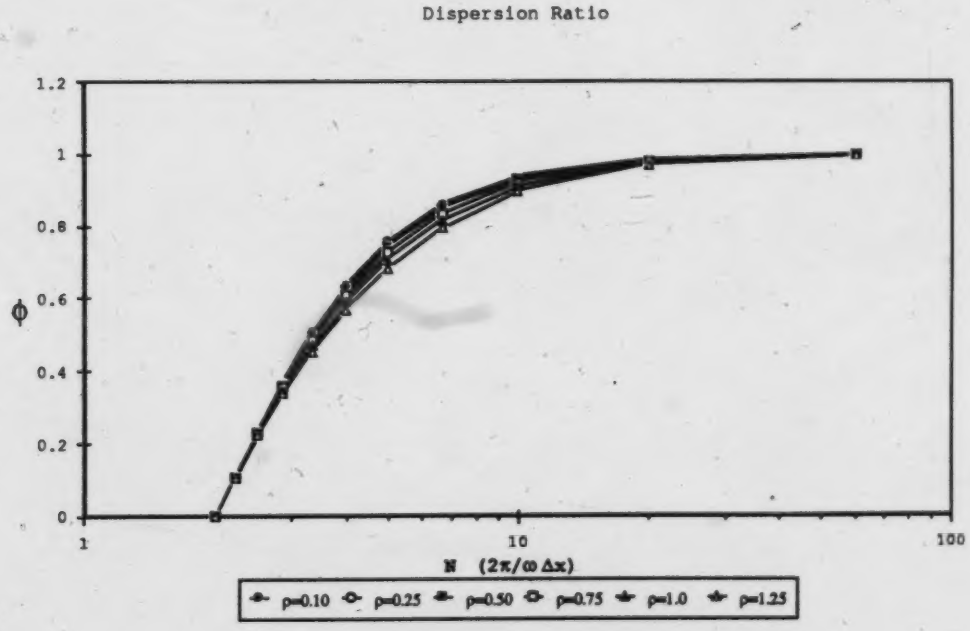


Figure 2.6 Dissipation and Dispersion for Central Difference

The effect of the excessive dispersion shown by the algorithm can be attested to the fact that as N approaches a value of 2, ϕ, which reflects the amount of dispersion, goes to zero. This implies that perturbations which can be defined by 2 nodes will not move at all. The result of this harsh restriction is clearly shown in Figure 2.7 which shows the results from using this algorithm for the advection of a gauss type wave. The propagated wave leaves a trail of oscillations behind due to this dispersion.

If a similar analysis is undertaken for a scheme which has some artificial diffusion, such as an algorithm based on an upwinded finite difference approximation for the advection term, the distortions shown by the central difference algorithm will disappear. The algorithm for such an upwinded approximation is

$$C_i^{n+1} + \rho\,\theta\left[C_i^{n+1} - C_{i-1}^{n+1}\right] = C_i^n - \rho\,(1-\theta)\left[C_i^n - C_{i-1}^n\right], \qquad (2.39)$$

and the corresponding terms for the von Neumann analysis that fit into Equations 2.38 and 2.39 are

$$\gamma_{real} = \left(1 - \rho\,(1-\theta)(1-\cos(\omega\Delta x))\right)\left(1 + \rho\,\theta(1-\cos(\omega\Delta x))\right) - \theta(1-\theta)\left(\rho\sin(\omega\Delta x)\right)^2$$

$$\gamma_{imaginary} = \left(1 - \rho\,(1-\theta)(1-\cos(\omega\Delta x))\right)\theta\,\rho\left(-\sin(\omega\Delta x)\right) + \left(1 + \rho\,\theta(1-\cos(\omega\Delta x))\right)(1-\theta)\,\rho\left(-\sin(\omega\Delta x)\right)$$

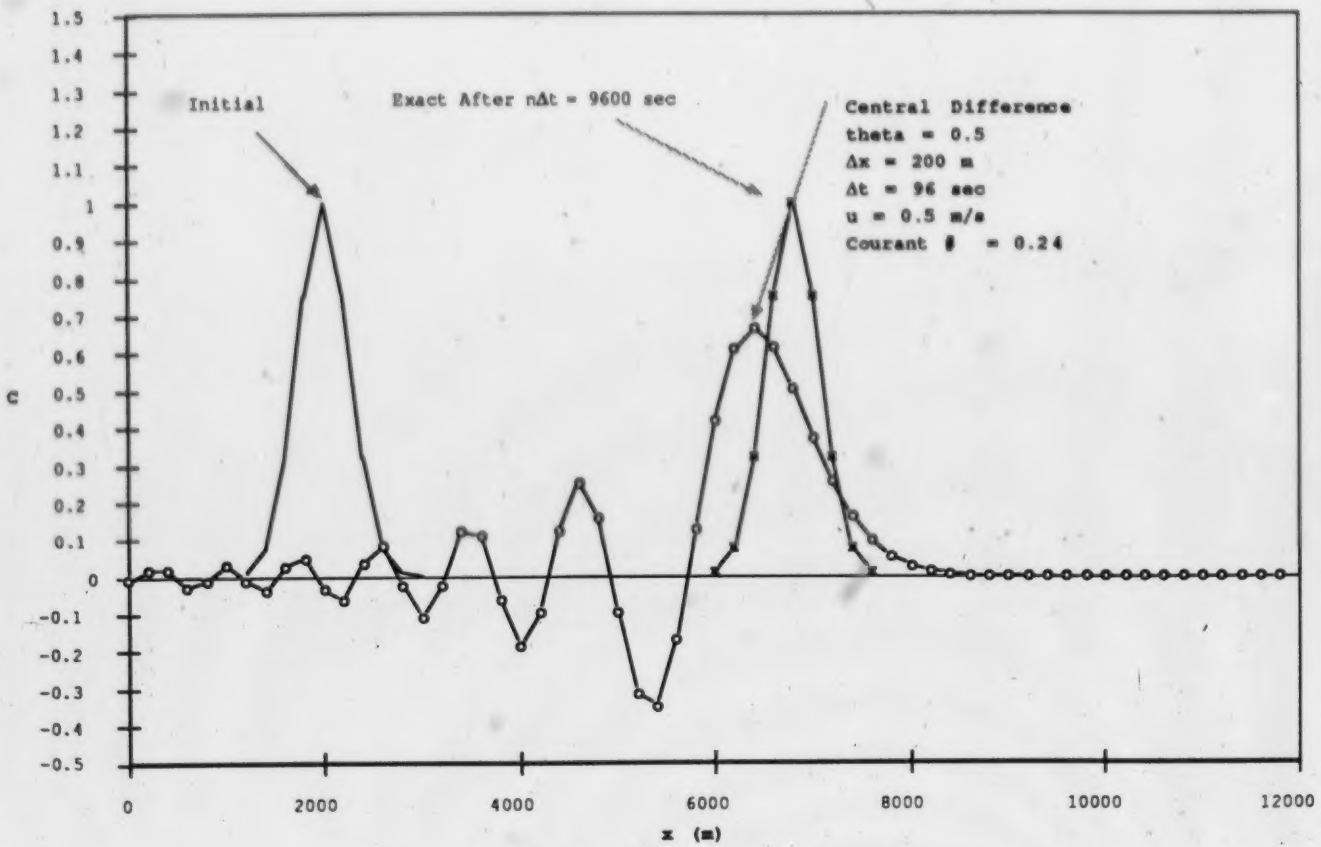


Figure 2.7 Advection of a Pulse using Central Difference

and

$$\gamma_{denominator} = \left(1 + \rho\,\theta\,(1-\cos(\omega\Delta x))\right)^2 + \left(\theta\,\rho\,\sin(\omega\Delta x)\right)^2 .$$

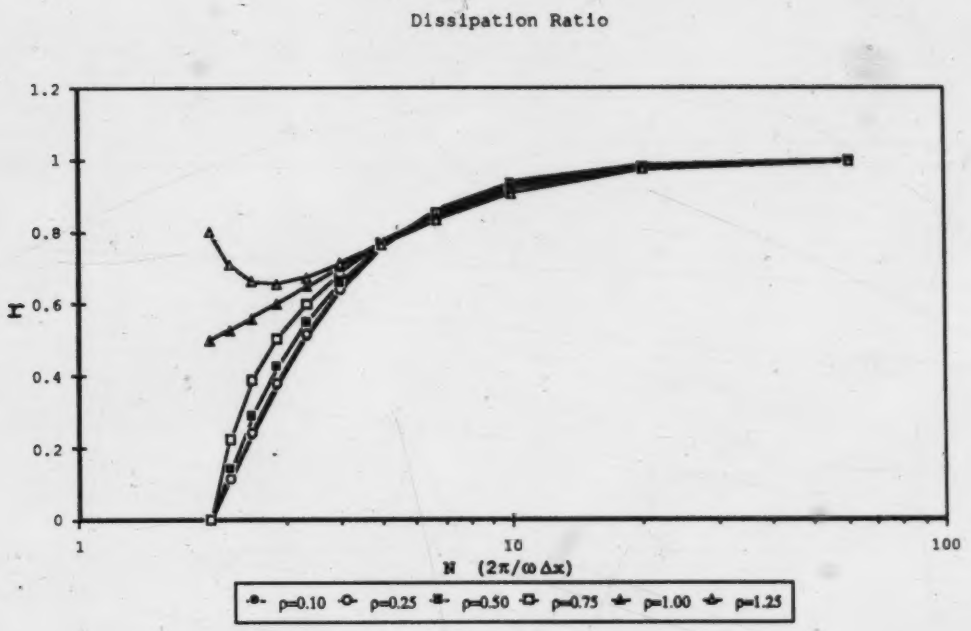
Figure 2.8 shows the plots for dissipation and dispersion the case when $\theta = 0.5$, a Crank Nicholson algorithm. The difference approximation has introduced artificial diffusion as illustrated by the fact that the dissipation(r) is less than one. However for cases other than $\rho = 1.0$, the algorithm shows dispersion greater than zero. This implies that any perturbations that may exist at the low frequencies will be dissipated as they are being advected. The dissipation produced by the algorithm can be seen in Figure 2.9 which shows the advection of a gauss wave at a $\rho = 0.24$ whose peak is reduced by sixty percent.

2.2.2.1.2 Taylor Series Analysis

The stability analysis discussed above shows that the upwinded schemes introduce dissipation in the solution. However to stipulate that this dissipation is in the form of diffusion, a further analysis must be undertaken. By looking at the Taylor Series representation of the discrete form, based on a general expression such as

$$C_{i+1}^n = C_i^n + \Delta x \frac{\partial C}{\partial x} + \frac{\Delta x^2}{2}\frac{\partial^2 C}{\partial x^2} + O(\Delta x)^3 + \ldots, \qquad (2.40)$$

the actual equation modelled by the discrete form can be deduced. The Taylor series representation of the upwinded



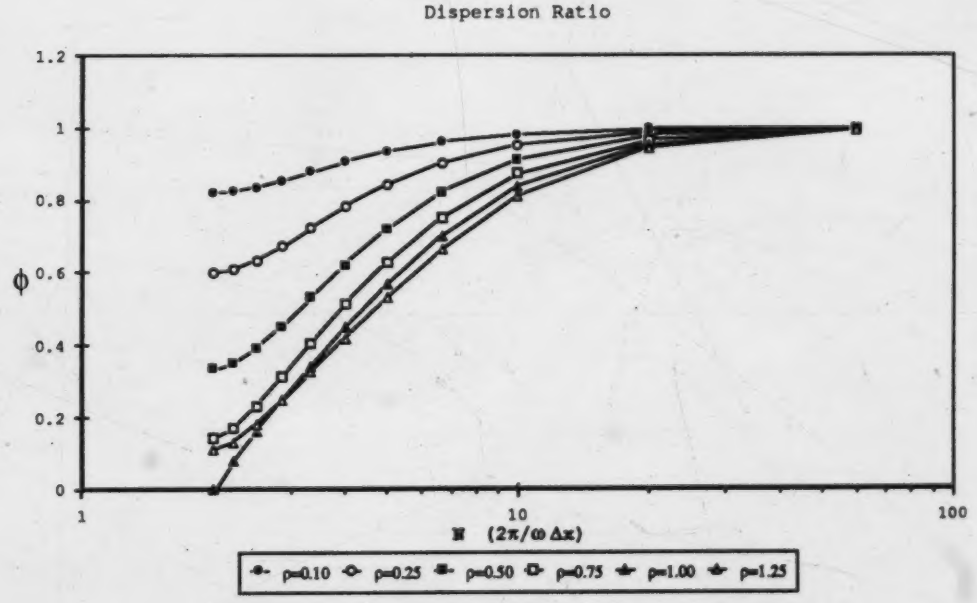


Figure 2.8 Dissipation and Dispersion for Upwinding Finite Difference

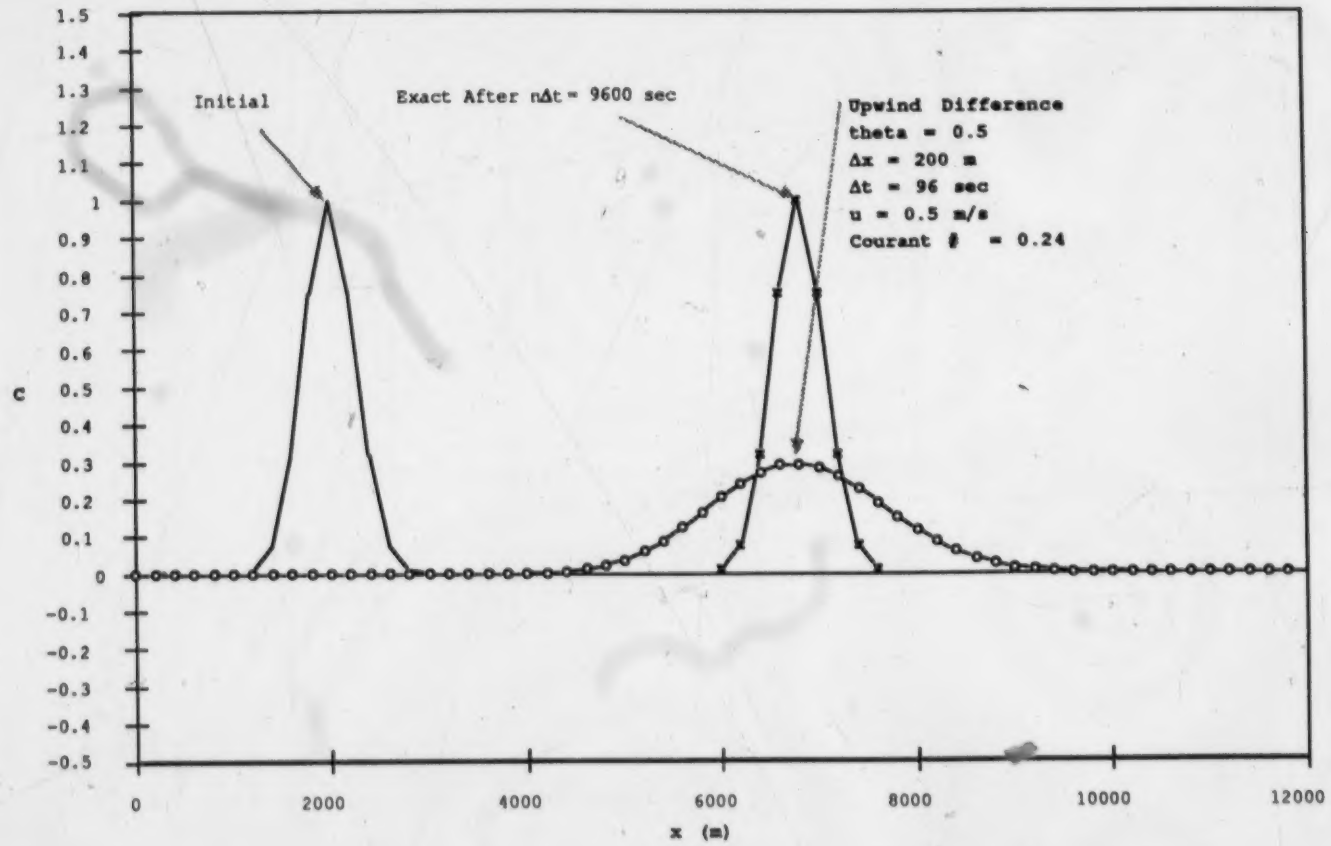


Figure 2.9 Advection of a Pulse using Upwinding FD

finite difference scheme, as given by Equation 2.39, shows that the equation modelled is actually

$$\frac{\partial C}{\partial t} + U\frac{\partial C}{\partial x} = \frac{U\Delta x}{2}\frac{\partial^2 C}{\partial x^2} + U\,(\Delta x)^2\frac{\partial^3 C}{\partial x^3} \; . \qquad (2.41)$$

From this equation it is evident that the modelled equation has an additional diffusion term which is dependent on the velocity and the discretization. Though this upwinded scheme shows that it has diffusion $\left(\text{ie } \frac{\partial^2 C}{\partial x^2}\right)$, upwinding schemes based on higher order approximations don't necessarily exhibit this artificial diffusion.

2.2.2.2 Popular Finite Difference Schemes

Many finite difference schemes that have been developed to overcome the difficulties presented by the advection diffusion equation. This section will discuss two such schemes that have been used in the analysis of the pollutant conservation equation as applied to natural rivers. One method that has been used in highly recognized programs, such as TRSMIX(G. Putz, 1984) and RIVMIX(Krishnappan and Lau, 1983), is the method developed by Stone and Brian(1963). Another method developed recently that has been used to simulate pollutant discharges through a network of rivers is the Holly-Preissmann method (Sauvaget 1985). Though these methods were used to solve two different problems, namely steady state transverse diffusion and unsteady longitudinal

dispersion, they can be shown to be solving the same differential equation, with the aid of a transformation.

2.2.2.2.1 Stone and Brian Method

The method developed by Stone and Brian(1963) has been used to solve the problem of transverse diffusion of a river plume. Generally the process of transverse diffusion in a steady plume as cast into the depth averaged form for a curvilinear coordinate system is given by the equation

$$\frac{\partial}{\partial x}(m_z UC) + \frac{\partial}{\partial z}(m_x VC) = \frac{\partial}{\partial x}\left(\frac{m_z}{m_x} E_x \frac{\partial C}{\partial x}\right) + \frac{\partial}{\partial z}\left(\frac{m_x}{m_z} E_z \frac{\partial C}{\partial z}\right). \quad (2.42)$$

Since most cases involving field measurements of velocities, the V velocity is impossible to measure and the effects of longitudinal diffusion are only important over very long reaches, these terms are neglected in the analysis. The simplified equation as derived in detail by Krishnappan and Lau(1983) is then stated as

$$\frac{\partial}{\partial x}(m_z UC) = \frac{\partial}{\partial z}\left(\frac{m_x}{m_z} E_z \frac{\partial C}{\partial z}\right). \quad (2.43)$$

Investigators such as G. Putz(1984) and Krishnappan and Lau(1983) have gone on to introduce a coordinate transformations which results in the equation

$$\frac{\partial C}{\partial x} + V \frac{\partial C}{\partial \eta} = D \frac{\partial^2 C}{\partial \eta^2} \quad (2.44)$$

where

$$\eta = \frac{q_c}{Q} = \frac{1}{Q} \int_0^z m_z \, h \, U \, dz \quad (2.45)$$

$$V = \frac{1}{Q^2}\frac{\partial}{\partial \eta}\left(U\ h^2\ m_x\ E_z\right) \quad \text{and} \tag{2.46}$$

$$D = \left(\frac{U\ h^2\ m_x\ E_z}{Q^2}\right). \tag{2.47}$$

Equation 2.44 is now in the advection diffusion form, which is subsequently solved using the numerical algorithm presented by Stone and Brian.

To develop a sufficiently accurate algorithm for Equation 2.44 Stone and Brian employed a general difference equation which is dependent on weighting coefficients. The difference equation developed was

$$\frac{1}{\Delta x}\left[g(C_{i+1,j} - C_{i,j}) + \frac{\theta}{2}(C_{i+1,j-1} - C_{i,j-1}) + m(C_{i+1,j+1} - C_{i,j+1})\right] +$$

$$\frac{V_{i,j}}{\Delta \eta}\left[a(C_{i,j+1} - C_{i,j}) + \frac{\varepsilon}{2}(C_{i,j} - C_{i,j-1}) + b(C_{i+1,j+1} - C_{i+1,j}) +\right.$$

$$\left. d(C_{i+1,j} - C_{i+1,j-1})\right] = \frac{D_{i,j}}{2\Delta \eta^2}\left[(C_{i,j+1} - 2C_{i,j} + C_{i,j-1}) +\right.$$

$$\left.(C_{i+1,j+1} - 2C_{i+1,j} + C_{i+1,j-1})\right] \tag{2.48}$$

where the coefficients a, $\frac{\varepsilon}{2}$, b, and d are weighting coefficients to evaluate the derivative $\frac{\partial c}{\partial \eta}$ and g, $\frac{\theta}{2}$, and m are for evaluating the 'time' derivative $\frac{\partial c}{\partial x}$. For a meaningful approximation the coefficients have to satisfy the conditions

$$a + \frac{\varepsilon}{2} + b + d = 1.0 \quad \text{and} \tag{2.49}$$

$$g + \frac{\theta}{2} + m = 1.0. \tag{2.50}$$

The diffusion term $\frac{\partial^2 C}{\partial \eta^2}$ is approximated using a Crank-Nicholson difference formula.

Stone and Brian go through a series of analyses, including the von Neumann Stability analysis, to arrive at optimum values for the coefficients:

$$g = \frac{2}{3};\ m = \frac{\theta}{2} = \frac{1}{6};\ a = b = d = \frac{\varepsilon}{2} = \frac{1}{4}\ . \qquad (2.51)$$

The resulting simplified difference equation from these optimum coefficients is

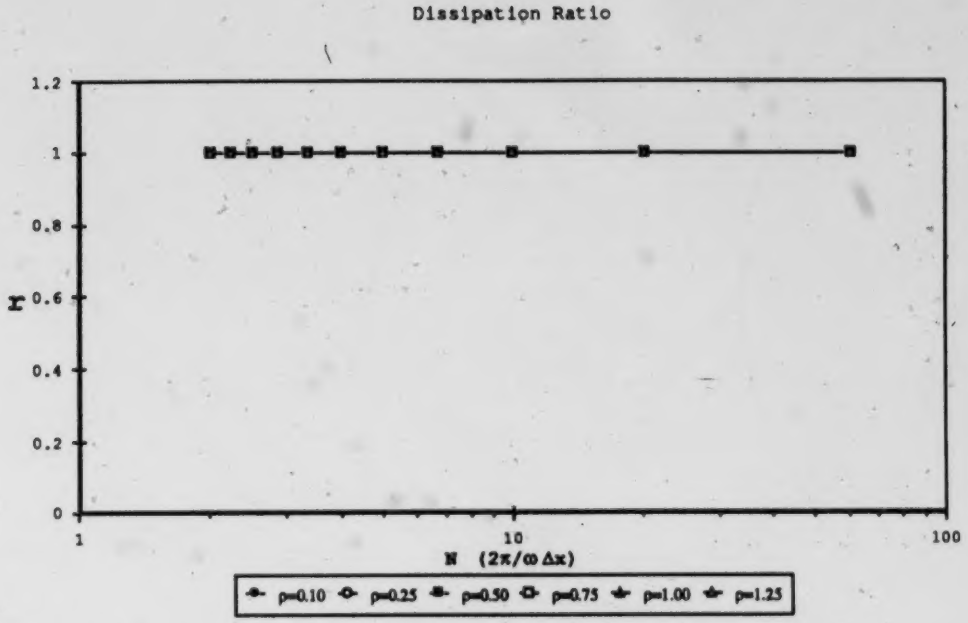
$$C_{i+1,j-1}\left(\frac{1}{6\rho'} - \frac{1}{4} - \frac{1}{2Pe}\right) + C_{i+1,j}\left(\frac{2}{3\rho'} + \frac{1}{2Pe}\right) + C_{i+1,j+1}\left(\frac{1}{6\rho'} + \frac{1}{4} - \frac{1}{2Pe}\right)$$

$$+ C_{i,j-1}\left(\frac{-1}{6\rho'} - \frac{1}{4} - \frac{1}{2Pe}\right) + C_{i,j}\left(\frac{-2}{3\rho'} + \frac{1}{2Pe}\right) + C_{i,j+1}\left(\frac{-1}{6\rho'} + \frac{1}{4} - \frac{1}{2Pe}\right)$$

$$= 0.0 \qquad (2.52)$$

where ρ' is a numerical Courant number define as $\frac{V_i\ \Delta x}{\Delta \eta}$ and Pe is the Peclet number defined as $\frac{V_{i,j}\Delta x}{D}$.

Figure 2.10 shows the resulting plots for the dissipation and dispersion resulting from the von Neumann stability analysis. This algorithm, like the centered difference algorithm, shows no dissipation and a similar dispersion behavior. The solution of the advection of a gauss shaped wave is shown in Figure 2.11 for a ρ' of 0.24. Though this solution shows some oscillations, they are a great deal smaller than those observed for centered



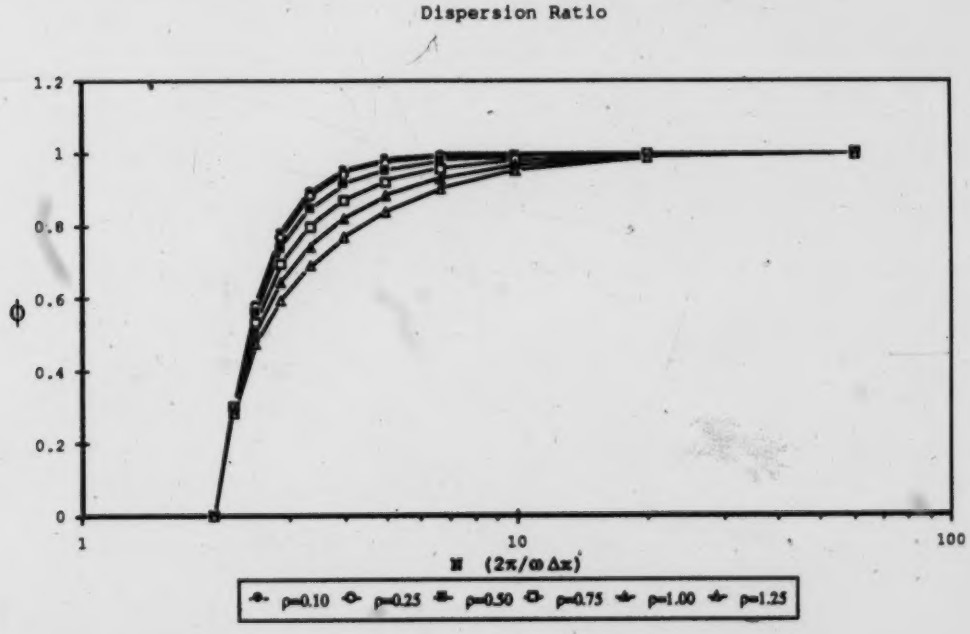


Figure 2.10 Dissipation and Dispersion for Stone and Brian

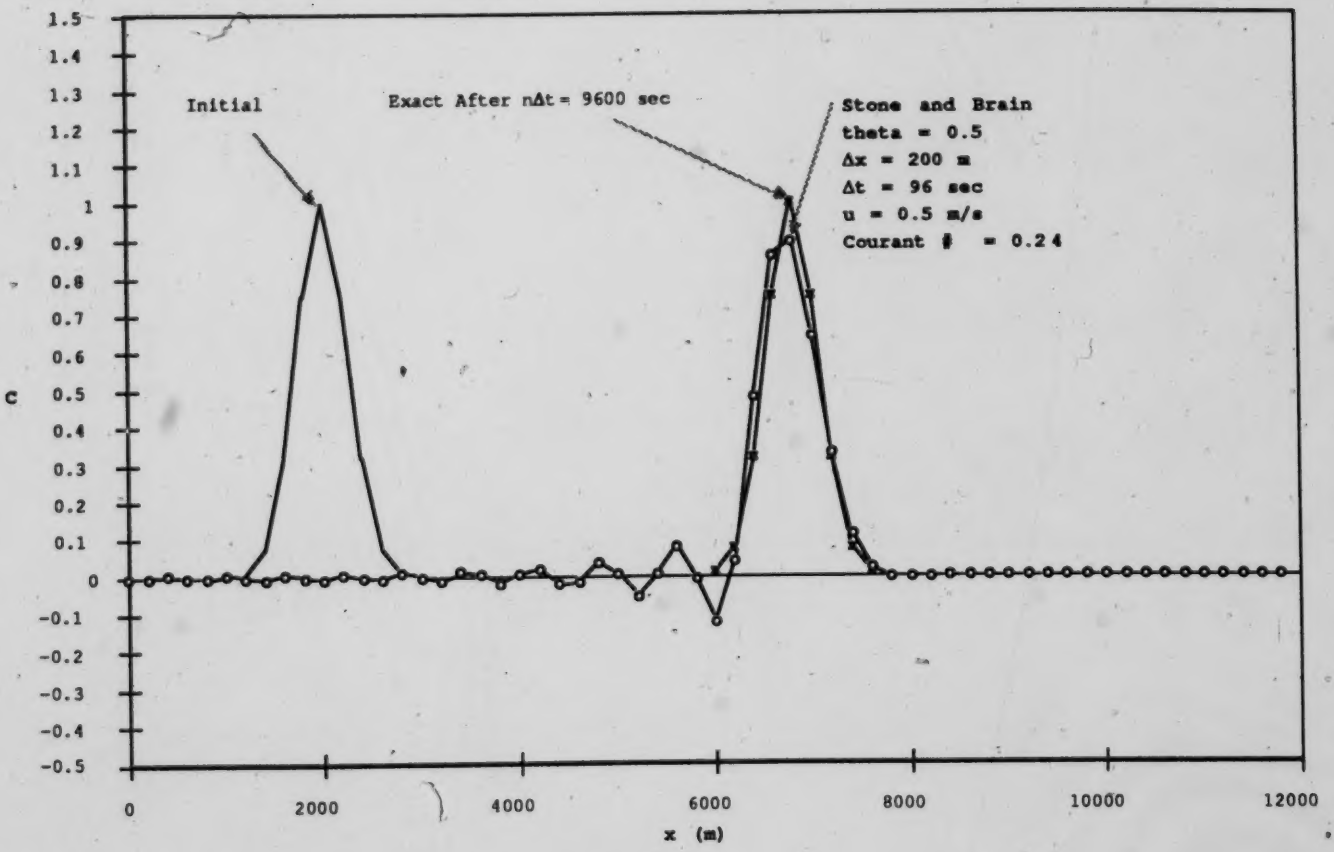


Figure 2.11 Advection of a Pulse using Stone and Brian

difference. A Taylor series analysis shows that this implicit algorithm models the equation

$$\frac{\partial C}{\partial x} + V\frac{\partial C}{\partial \eta} = D\frac{\partial^2 C}{\partial \eta^2} + O(\Delta\eta)^2 + O(\Delta x)^2. \qquad (2.53)$$

This algorithm is satisfactory in most applications as the effective Courant number $\left(\frac{V\Delta x}{\Delta \eta}\right)$ used is very small.

2.2.2.2.2 The Holly-Preissmann Method

One of the newer methods to be applied to the solution of pollutant conservation in river systems is the Holly-Preissmann method (Sauvaget 1985). Basically their method involves the use of Hermite Polynomials to interpolate concentrations and concentration gradients, which are subsequently advected by the Characteristic method to the new time step. Due to the iterative nature of the solution algorithm, a simplified version for the case of constant velocity field was used to solve for the advection of the one dimensional pulse.

The results of this test is shown in Figure 2.12 for a $\rho = 0.24$, as was done for the other methods. The plot shows that though the pulse is advected at the right speed, there does appears to be some dissipation of the peak. There is also the persistence of a pair of small wiggles on either side attached to the pulse. These observations tend to indicate that though the solution algorithm shows very little dispersion, the high frequency disturbances still produce

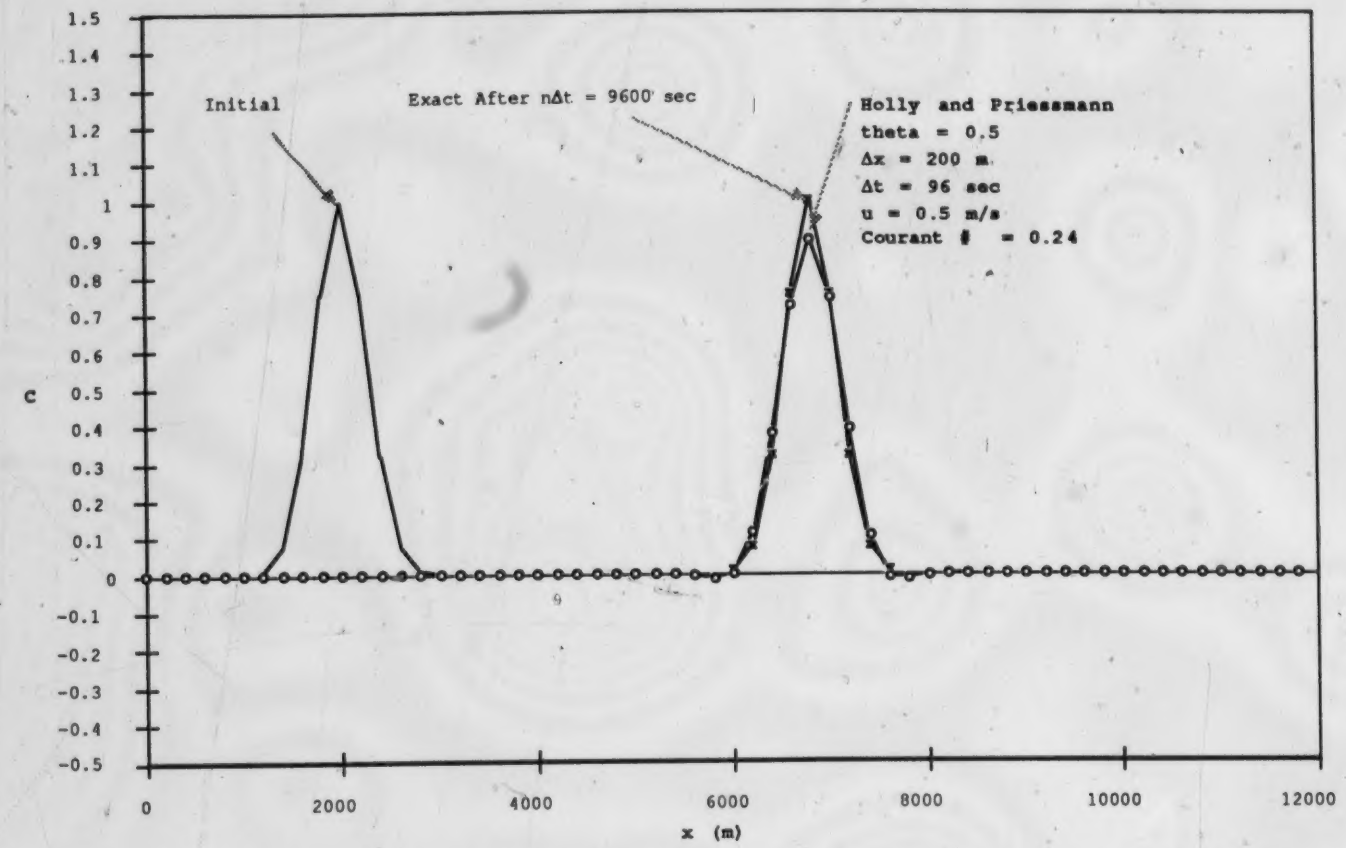


Figure 2.12 Advection of a Pulse using Holly and Preissmann

oscillations, even at this low a Courant number(0.24). There also appears to be some dissipation of the peak.

2.2.2.3 Finite Element Methods

Generally the finite element method has been restricted to use by the mathematicians due largely to its complexity. However recently a number of investigators have applied the Finite Element formulation to solve basic hydraulic problems. One of the first to apply this method to the conservation equation was Leikuhler *et al.* (1974). The formulation of the finite element consisted of the Method of Weighted Residuals using triangular elements. These linear triangular elements, at $\theta = 0.5$, give exactly the same results as for one dimensional advection of a gauss type wave using the Stone and Brian method discussed earlier. The results from their study indicate that the prospect of getting a viable solution using the Finite Element method seem to be very good.

3. FINITE ELEMENT METHOD

The advancement of Computational Fluid Mechanics is hindered by two basic restrictions. One limitation is the computing power available to perform the calculations and the other is the accuracy of the method, which is related to its complexity. Modelling fluid flow for practical applications requires that a balance be achieved between the degree of discretization and the complexity of the algorithm used. Due to these limitations the modeler has to develop new and generally more complicated models to accurately model the behavior in question.

The hydraulic engineer is further faced with the fundamental problem of irregular boundaries which do not conform to any orthogonal coordinate system. A method that has the ability to overcome the limitations of boundary fitted coordinate systems is the Finite Element Method (FEM). In the Finite Element Method the domain of a problem is discretized into a number of elements that can be of varying shape, size, orientation, and numerical accuracy. The discretized domain is subsequently formulated into a matrix which is solved for the desired unknowns.

The following is a general introduction to the FEM as applied to the depth averaged conservation equation (Equation 2.16). An introduction to the use of a new type of upwinding elements is presented. The flow chart of the program developed to implement these new elements is also discussed.

3.1 Method of Weighted Residuals

A literature review of the Finite Element Methods would realize a variety of different formulations that all result in the discretization of the domain of the problem into finite elements. Due to its simplicity and elegance, the general formulation of the method of Weighted Residuals was used for the present investigation.

Basically the premise of any finite element method as related to solving differential equations is to reduce the constraints of the problem so that a set of simple algebraic functions can describe the solution over a discrete element.

3.1.1 The Weak Statement

The first step in the Method of Weighted Residuals is to derive the weak statement of the governing equation. The method requires that the solution should be approximated over the finite element by a set of basis functions, so that

$$C(t,x,z) \cong \sum_{j=1}^{N} C_j(t)\ f_j(x,z), \qquad (3.1)$$

where $C_j(t)$ are the nodal values of the concentration as a function of time, $f_j(x,z)$ are a set of basis functions, and N is the number of unknowns.

If the basis functions are substituted into the original partial differential equation the result would be that a residual will exist. This residual would represent the error due to the approximate solution not being the exact solution.

The residual is weighted to a local domain specific to each element by multiplying the residual by a set of weight or test functions. This weight residual is then integrated over the domain to represent an average residual over the whole domain. The resulting integral can be stated as,

$$\int_{\Omega} v_i(x,z)\ L(C)\ d\Omega = \text{Global Residual} \qquad (3.2)$$

where $L(C)$ is the partial differential equation for the discrete representation given by 3.1, $v_i(x,z)$ are the test functions and Ω is the domain of the problem. The equations needed to solve for the discrete solution C_j are generated by forcing the Global Residual to be zero.

The advantage of such a formulation is its ability to reduce the restrictions on the basis functions. The original differential equation would generally have a second derivative. This would imply that for the basis functions to represent the solution, even over a single element, they would have to have continuous first derivative, meaning that the second derivative exists. This restriction would eliminate the use of simple linear functions, since they have a discontinuous first derivative. However with the use of a simplified weak statement, this restriction can be relaxed so as to allow broad range of possible functions.

The resulting integral equation 3.2 can be simplified by using the Green-Gauss theorem to remove this restrictions. The general Green-Gauss theorem can be stated as,

$$\int_{\Omega} \beta_i \frac{\partial \omega_k}{\partial x_j} \, d\Omega = \int_{\Gamma} \beta_i (\omega_k \circ n_j) \, d\Gamma - \int_{\Omega} \frac{\partial \beta_i}{\partial x_j} \circ \omega_k \, d\Omega \quad (3.3)$$

where β_i are a set of scalar functions, ω_k are the basis functions, n_j is the normal vector, Ω is the domain of the problem, and Γ is the boundary portion of the domain.

The resulting simplified statement is termed the weak statement since the solution restrictions of the weak statement are less than those of the original differential equation. Originally the solution domain required that a second derivative of the functional of the solution must exist. The new statement requires that the first derivative of the functional need only be square integratable. Thus the solution domain of the original equation is a sub-set of the solution domain of the weak statement.

Applying this technique to the depth averaged concentration equation initially results in the integral equation shown below,

$$\int_{\Omega_{x_i}} \left(\underset{(I)}{v_i \frac{\partial C}{\partial t}} + \underset{(II)}{v_i U_i \frac{\partial C}{\partial x_i}} - \underset{(III)}{v_i \frac{\partial}{\partial x_i} E_{ij} \frac{\partial C}{\partial x_j}} - \underset{(IV)}{v_i P} \right) dx_i = 0 \quad (3.4)$$

where v_i are the test functions, and Ω_{x_i} is the domain in the two directions. By applying the Green-Gauss theorem to the third term, it can be simplified to the equation shown below,

$$- \int_{\Omega_{x_i}} v_i \frac{\partial}{\partial x_i} E_{ij} \frac{\partial C}{\partial x_j} \, dx_i = - \int_{\Gamma_{x_i}} v_i (E_{ij} \frac{\partial C}{\partial x_j} \circ n_j) \, d\Gamma$$

$$+ \iint_{\Omega_{x_i}} \frac{\partial v_i}{\partial x_j} \circ E_{ij} \frac{\partial C}{\partial x_j} \, dx_i \, . \qquad (3.5)$$

The boundary integral in the above equation is termed the natural boundary condition. The reason it is natural is that the flux boundary conditions can be incorporated in the original weak statement automatically. Thus there is no need to use any extra formulation for flux boundaries.

The simplified weak statement of Equation 2.16 is

$$\iint_{\Omega_{x_i}} \left(v_i \frac{\partial C}{\partial t} + v_i U_i \frac{\partial C}{\partial x_i} + \frac{\partial v_i}{\partial x_j} \circ E_{ij} \frac{\partial C}{\partial x_i} - v_i P \right) dx_i$$

$$- \int_{\Gamma_{x_i}} v_i (E_{ij} \frac{\partial C}{\partial x_i} \circ n_j) \, d\Gamma = 0 \qquad (3.6)$$

3.1.2 Semi-Discrete Form

The discrete representation expressed by the nodal values and the basis functions are substituted into the weak statement to derive the semi-discrete form of the equation shown below,

$$\int_{\Omega_x} \int_{\Omega_z} \Big[v_i f_j \frac{\partial C_j}{\partial t} + v_i \left(C_j U_j \frac{\partial f_j}{\partial x} + C_j V_j \frac{\partial f_j}{\partial z} \right)$$

$$+ \frac{\partial v_i}{\partial x} \left((E_{xx})_j C_j \frac{\partial f_j}{\partial x} + (E_{xz})_j C_j \frac{\partial f_j}{\partial z} \right)$$

$$+ \frac{\partial v_i}{\partial z} \left((E_{zx})_j C_j \frac{\partial f_j}{\partial x} + (E_{zz})_j C_j \frac{\partial f_j}{\partial z} \right) + v_i P_j \Big] \, dz dx. \qquad (3.7)$$

This statement can be further simplified by performing the integral in two parts. The first integrate over each

element, then to sum the element integrals over the whole domain. Each integral in the statement has an associated matrix as classified below,

the mass matrix:

$$[M] = \int_{\Omega_x}\int_{\Omega_z} v_i \, f_j \; dzdx;$$

the stiffness matrix:

$$[K] = \int_{\Omega_x}\int_{\Omega_z} v_i \left(U_j \frac{\partial f_j}{\partial x} + V_j \frac{\partial f_j}{\partial z} \right) + \frac{\partial v_i}{\partial x}\left((E_{xx})_j \frac{\partial f_j}{\partial x} + (E_{xz})_j \frac{\partial f_j}{\partial z} \right) + \frac{\partial v_i}{\partial z}\left((E_{zx})_j \frac{\partial f_j}{\partial x} + (E_{zz})_j \frac{\partial f_j}{\partial z} \right) dzdx \; ;$$

and the force matrix: $[F] = \int_{\Omega_x}\int_{\Omega_z} v_i \, P_j \; dzdx.$

Thus the semi-discrete form can be simply expressed as

$$[M]\left\{\frac{dC_j}{dt}\right\} + [K]\{C_j\} + [F] = \{0\}. \qquad (3.8)$$

3.1.3 Coordinate Transformation

The finite element method has the advantage of being able to incorporate a mapping system into its basic formulation. In a general case, any arbitrary shape of domain can be discretized into a number of elements of different shape and size. Each element in the domain has to be mapped into a simple normalized local domain of a general elements, shown in Figure 3.1.

The mapping is accomplished by using a set of parametric functions such that the coordinates can be expressed as,

$$x = \sum_{i=0}^{Nn} X_i \, g_i(r,s) \quad \text{and} \quad z = \sum_{i=0}^{Nn} Z_i \, g_i(r,s).$$

In these expressions X_i and Z_i are nodal coordinates, r and s represent the local coordinate system, g_i are a set of geometric transformation functions, and Nn is the number of nodes in the element.

The present formulation uses basis functions as the geometric functions. Using the above mapping method, the transformation from global coordinate system to local system is expressed by the relationship,

$$\begin{Bmatrix} \frac{\partial f_i}{\partial r} \\ \frac{\partial f_i}{\partial s} \end{Bmatrix} = \underbrace{\begin{bmatrix} \frac{\partial f_i}{\partial r} X_j & \frac{\partial f_i}{\partial r} Z_j \\ \frac{\partial f_i}{\partial s} X_j & \frac{\partial f_i}{\partial s} Z_j \end{bmatrix}}_{[J]} \begin{Bmatrix} \frac{\partial f_i}{\partial x} \\ \frac{\partial f_i}{\partial z} \end{Bmatrix} \tag{3.9}$$

However the transformation needed is usually the reverse, that is from local to global, which can be determined by the transformation expression shown below,

$$\begin{Bmatrix} \frac{\partial f_i}{\partial x} \\ \frac{\partial f_i}{\partial z} \end{Bmatrix} = [J]^{-1} \begin{Bmatrix} \frac{\partial f_i}{\partial r} \\ \frac{\partial f_i}{\partial s} \end{Bmatrix} \tag{3.10}$$

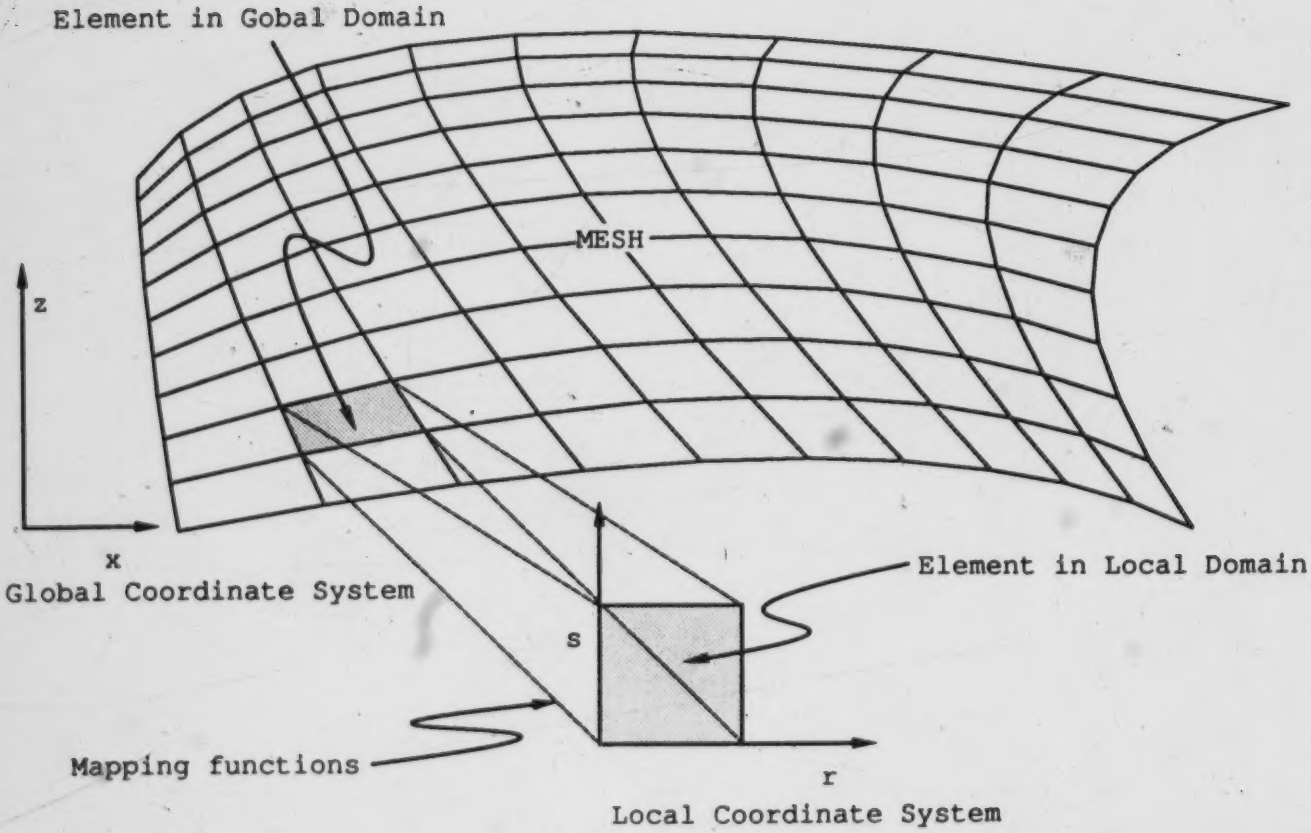


Figure 3.1 Coordinate Transformation Using Mapping Functions

3.1.4 Time Discretization

There are a number of different methods available to discretize the unsteady term in the partial differential equation. The most commonly used is the θ implicit method.

The θ implicit method applied to Equation 3.8 results in the discrete equation shown below,

$$[M]\left\{C_i^{n+1}\right\} - [M]\left\{C_i^{n}\right\} +$$

$$\theta\Delta t\ [K]\left\{C_i^{n+1}\right\} + (1-\theta)\Delta t\ [K]\left\{C_i^{n}\right\} + \Delta t\{F\} = \{0\} \qquad (3.11)$$

3.1.5 Boundary Conditions

The current thesis incorporates in its formulation a general method whereby a variety of boundary condition can be specified on any arbitrary boundary.

To incorporate any general boundary condition the boundary integral resulting from the integration by parts of the second derivative was substituted with an arbitrary penalty expression, as show below,

$$\frac{\partial C}{\partial n} = A\ C + B\ . \qquad (3.12)$$

By adjusting the values of the coefficients A and B, one can provide for a fixed flux boundary condition, fixed value boundary condition, or a combination of both.

The resulting expressions substituted into the weak form are

$$\int_{\Gamma} v_i C_j \frac{\partial f_i}{\partial n}\, d\Gamma = \int_{\Gamma} \left(A_j\, v_i C_j f_j - v_i\, B_j \right)\, d\Gamma = 0. \qquad (3.13)$$

In this equation *n* indicates that only the portion that is normal to the boundary of the integral survives.

3.1.6 Discrete Equation

In summary the discrete equation, for each element, coded into a program can be expressed as the following:

$$[M]\left\{C_i^{n+1} - C_i^n\right\} + \theta\Delta t\left([K_x] + [K_z]\right)\left\{C_i^{n+1}\right\}$$

$$+ (1-\theta)\Delta t\left([K_x] + [K_z]\right)\left\{C_i^n\right\} + \Delta t\,\{F\}$$

$$+ \Delta t\,[K_B]\left\{C_i^{n+1}\right\} + \Delta t\,\{F_B\} = \{0\} \qquad (3.14)$$

where,

$$[M] = \int_{\Omega_r}\int_{\Omega_s} v_i\, f_j\, [J]^{-1} ds dr\ ;$$

$$[K_x] = \int_{\Omega_r}\int_{\Omega_s} v_i\, U_j\, [J_x]^{-1}\frac{\partial f_i}{\partial r} + [J_x]^{-1}\frac{\partial v_i}{\partial r}(E_{xx})_j [J_x]^{-1}\frac{\partial f_i}{\partial r}$$

$$+ [J_x]^{-1}\frac{\partial v_i}{\partial r}(E_{xz})_j [J_z]^{-1}\frac{\partial f_i}{\partial s}\,[J]^{-1}\, ds dr\ ;$$

$$[K_z] = \int_{\Omega_r}\int_{\Omega_s} v_i\, V_j\, [J_z]^{-1}\frac{\partial f_i}{\partial s} + [J_z]^{-1}\frac{\partial v_i}{\partial s}(E_{zx})_j [J_x]^{-1}\frac{\partial f_i}{\partial r}$$

$$+ [J_z]^{-1}\frac{\partial v_i}{\partial s}(E_{zz})_j\, [J_z]^{-1}\frac{\partial f_i}{\partial s}\,[J]^{-1}\, ds dr\ ;$$

$$\{F\} = \int_{\Omega_r}\int_{\Omega_s} v_i \; P_j \quad [J]^{-1} \, dsdr \; ;$$

$$[K_B] = \int_{\Gamma} A_j \; v_i f_j \, dn \quad \text{and}$$

$$\{F_B\} = \int_{\Gamma} v_i \; B_j \, dn \; .$$

In the above expressions the following convenience was used,

$$\begin{bmatrix} [J_x]^{-1} \\ [J_z]^{-1} \end{bmatrix} = \begin{bmatrix} [J_{xx}]^{-1} & [J_{xz}]^{-1} \\ [J_{zx}]^{-1} & [J_{zz}]^{-1} \end{bmatrix} \equiv [J]^{-1} \; .$$

3.1.7 The Solution Algorithm

The solution scheme for a general case of the finite element method proceeds as the following:

1) The weak statement is evaluated for each of the element using the appropriate set of functions (Equation 3.13);

2) The element matrices are assembled into a global matrix;

3) This global matrix of linear algebraic set of equations is solved using any particular method.

The global set of linear algebraic equations for the finite element formulation can be stated as,

$$[K']\left\{C_i^{n+1}\right\} = \{F'\}, \qquad (3.15)$$

in which,

$$[K'] = \sum_{i=1}^{N} \left([M] + \theta\Delta t \left([K_x]+[K_z] \right) \right),$$

$$\{F'\} = \sum_{i=1}^{N} \left((1-\theta)\Delta t \left([K_x]+[K_z] \right) \left\{C_i^n\right\} + \Delta t \{F\} \right)$$

$$+ \sum_{i=1}^{Nb} \left([K_B] \left\{C_i^n\right\} + \{F_B\} \right).$$

Where N is the number of elements, and Nb is the number of boundary elements in the problem.

Though the above algorithm is well behaved, it does require a large block of memory to execute, even though Sky-lined storage scheme was used. In addition, the limits put on the available computational effort forced the addition of the alternating direction formulation discussed in some detail by Baker (Baker,1983).

3.2 Choice of Test and Basis Functions

The solution algorithm developed to this point is still incomplete since the test and basis functions have not been chosen. In the finite element method the choice of these functions have been categorized into two distinct groups. When the choice of the test and basis functions is the same then the formulation is termed the Bubnov Galerkin formulation. However if the test and basis functions are different then the method is termed a Petrov Galerkin formulation.

The Bubnov Galerkin formulation results in a square matrix, and for the particular case when linear elements are used its behavior is similar to the centered finite difference formulations.

The Petrov Galerkin formulation has generally been based on the use of higher order upwinding functions as test functions and simpler functions as basis functions. The test functions are generally developed by weighting the basis functions in an upwind manner.

Recently however a paper by Steffler (1988) presented, the use of higher order upwind functions as the basis functions. The reasoning behind this approach is that for convection dominating variables, the value at a new point should have a heavier dependence on the preceding points than the points in its surrounding. The result of this formulation is a scheme that reflects the behavior observed using the popular QUICK finite volume schemes (Leonard, 1979).

The current investigation will have two major purposes in mind. One is the judge the behavior of these new upwinding basis functions in a series of numerical tests, and the other is to observe their behavior in some simple practical problems. To under take such an investigation three different levels of accuracy were tested for the numerical tests.

3.2.1 Choice of Basis and Test Functions

The numerical tests were based on the following choices for the test and basis functions,

1) **LBG** : Linear Bubnov Galerkin,

-uses linear test and basis function as stated below for the one dimensional case:

$$f(r) = \begin{Bmatrix} \frac{(1+r)}{2} \\ \frac{(1-r)}{2} \end{Bmatrix} \; ; \; r \in (-1,1) \qquad (3.16)$$

2) **QUPG** : Quadratic Upwinding Petrov Galerkin,

-uses quadratic basis functions as shown below for the one dimensional case, and the linear test functions 3.16,

$$f(r) = \begin{Bmatrix} \frac{(3+r)(1-r)}{4} \\ \frac{(1+r)(3+r)}{8} \\ \frac{-(1-r)(1+r)}{8} \end{Bmatrix} \; ; \; r \in (-1,1) \qquad (3.17)$$

3) **CUPG** : Cubic Upwinding Petrov Galerkin,

-uses cubic basis functions as shown below for the one dimensional case, and the linear test functions 3.16,

$$f(r) = \left\{ \begin{array}{c} \frac{(5+r)(3+r)(1-r)}{16} \\ \frac{(1+r)(3+r)(5+r)}{48} \\ \frac{-(1-r)(1+r)(5+r)}{16} \\ \frac{-(1-r)(1+r)(3+r)}{48} \end{array} \right\} \; ; \; r \in (-1,1) \quad . (3.18)$$

The surrounding elements that influence the general two dimensional element, as well as the specialized element near the boundaries, are shown for LBG, QUPG, and CUPG in Figures 3.2, 3.3 and 3.4 respectively.

3.2.2 The Program

The program was written originally by Dr. P. M. Steffler at the University of Alberta on an Apple Macintosh computer, using the C programming language. This version of the program was modified to include the solution for the depth average concentration in rivers and expanded to include the Cubic basis functions.

A general flow chart of the program is shown in Figures 3.5 to 3.9. This flow chart is meant to be a guide and aid in the use of the program in a user friendly environment. The flow chart indicates the flexibility incorporated into the program to solve problems posed in a variety of different ways. These range for the solution for plume in an idealized rectangular domain, to the laborious solution of a slug test in a general river reach.

Versions of the program were built for the Amdahl mainframe, IBM PC and Apple Macintosh machines. Two specialized versions of the program were developed, one general enough to use on any machine, and another based solely on the user friendly environment of the Apple Macintosh machine.

A listing of the general program is also included in Appendix 1. The program was written in the C language due to its ability to allow dynamic allocation of memory.

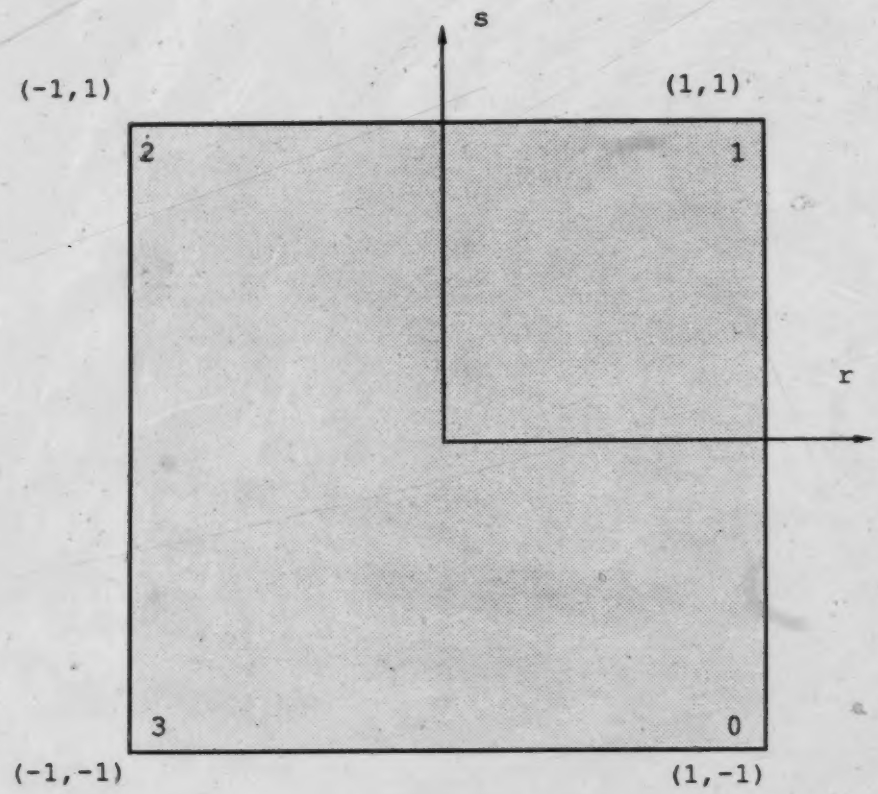
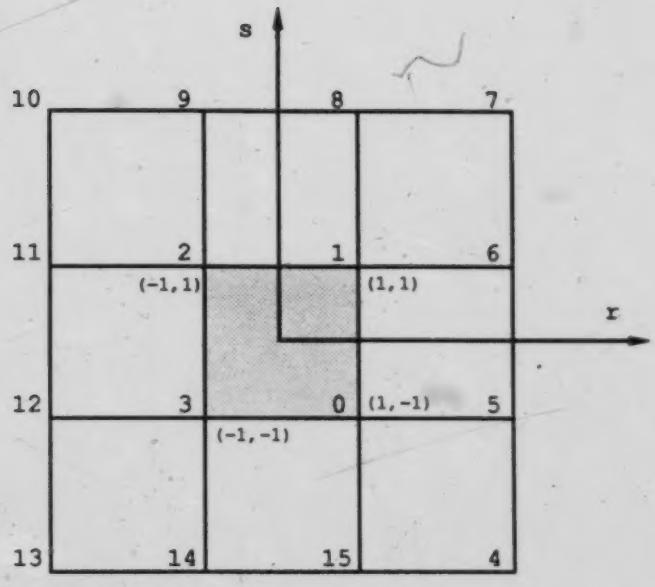
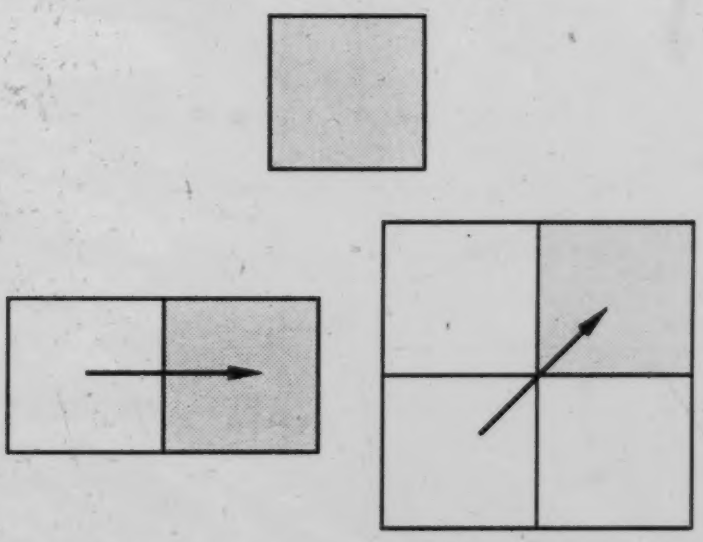


Figure 3.2 Local Domain of the Linear Rectangular Element

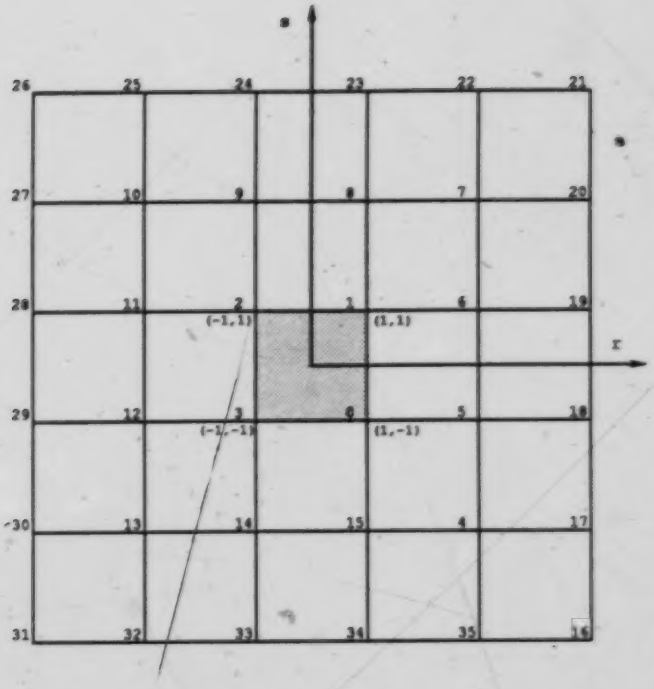


General area of influence for an element using Quadratic Upwinding Elements

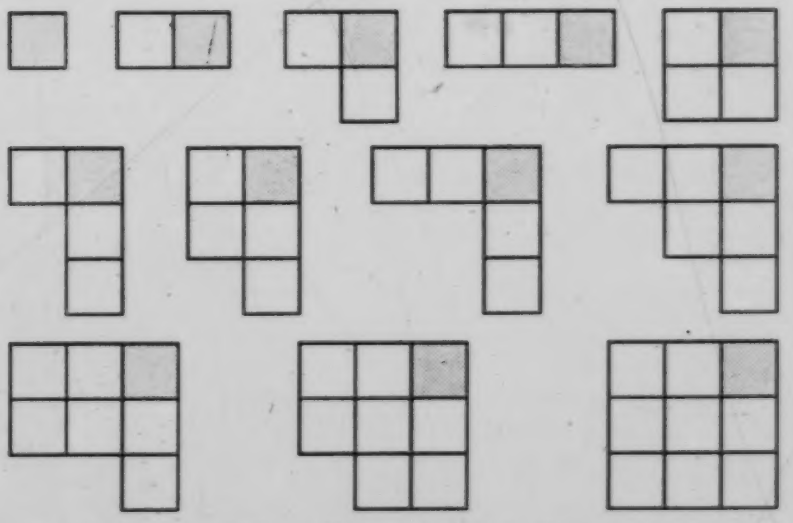


Subset of Possible Element Types for Quadratic Upwinding Elements

Figure 3.3 Local Domain of the Quadratic Upwinding Rectangular Element



General area of influence for an element using Cubic Upwinding Elements



Subset of Possible Element Types for Cubic Upwinding Elements

Figure 3.4 Local Domain of the Cubic Upwinding Rectangular Element

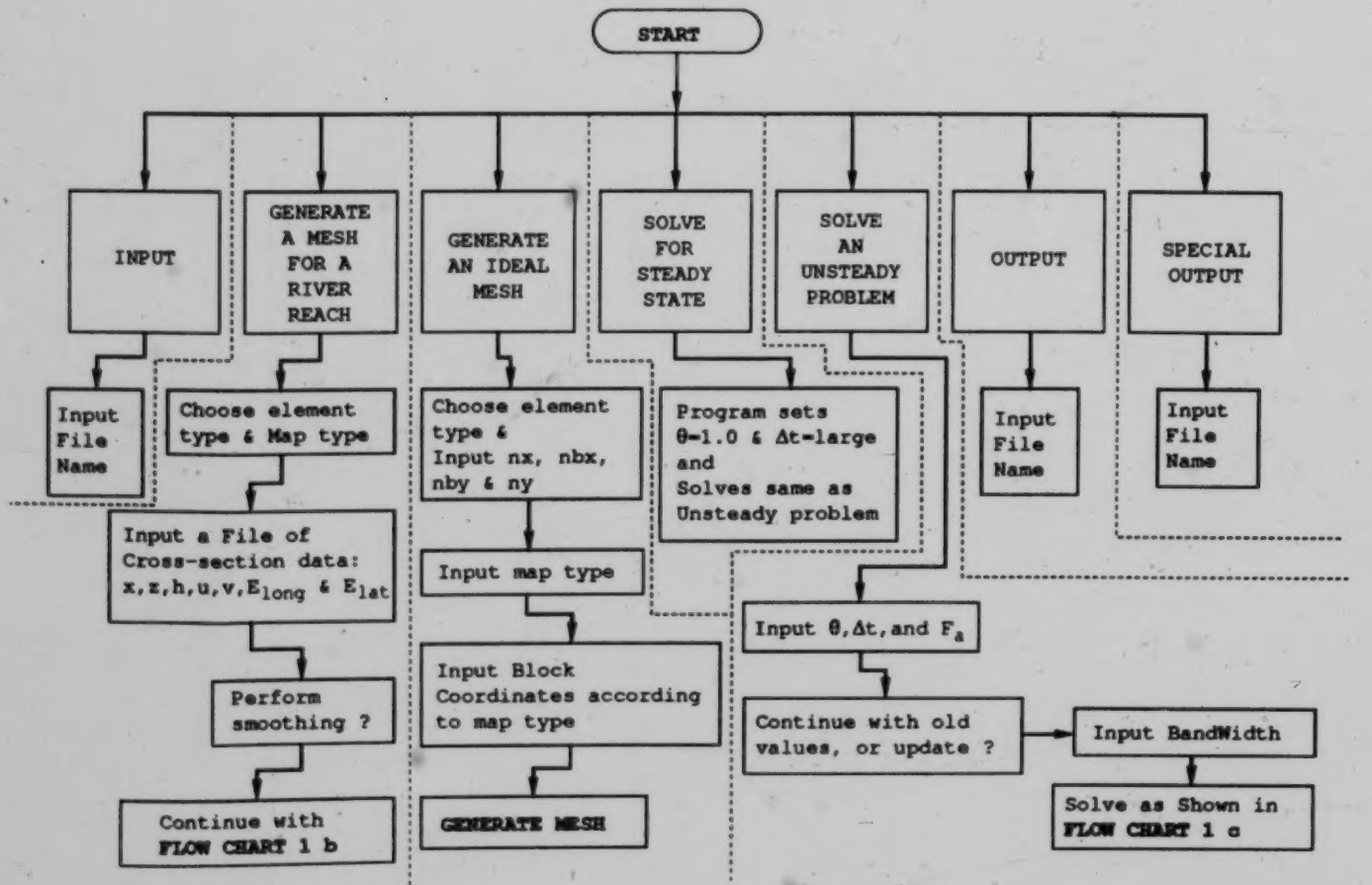

START
INPUT
GENERATE A MESH FOR A RIVER REACH
GENERATE AN IDEAL MESH
SOLVE FOR STEADY STATE
SOLVE AN UNSTEADY PROBLEM
OUTPUT
SPECIAL OUTPUT
Input File Name
Choose element type & Map type
Choose element type & Input nx, nbx, nby & ny
Program sets θ=1.0 & Δt=large and Solves same as Unsteady problem
Input File Name
Input File Name
Input a File of Cross-section data: x,z,h,u,v,E_{long} & E_{lat}
Input map type
Input θ,Δt,and F_a
Perform smoothing ?
Input Block Coordinates according to map type
Continue with old values, or update ?
Input BandWidth
Continue with FLOW CHART 1 b
GENERATE MESH
Solve as Shown in FLOW CHART 1 c

Figure 3.5 Flow Chart 1 a, A General Flow Chart of the Program CDFEM

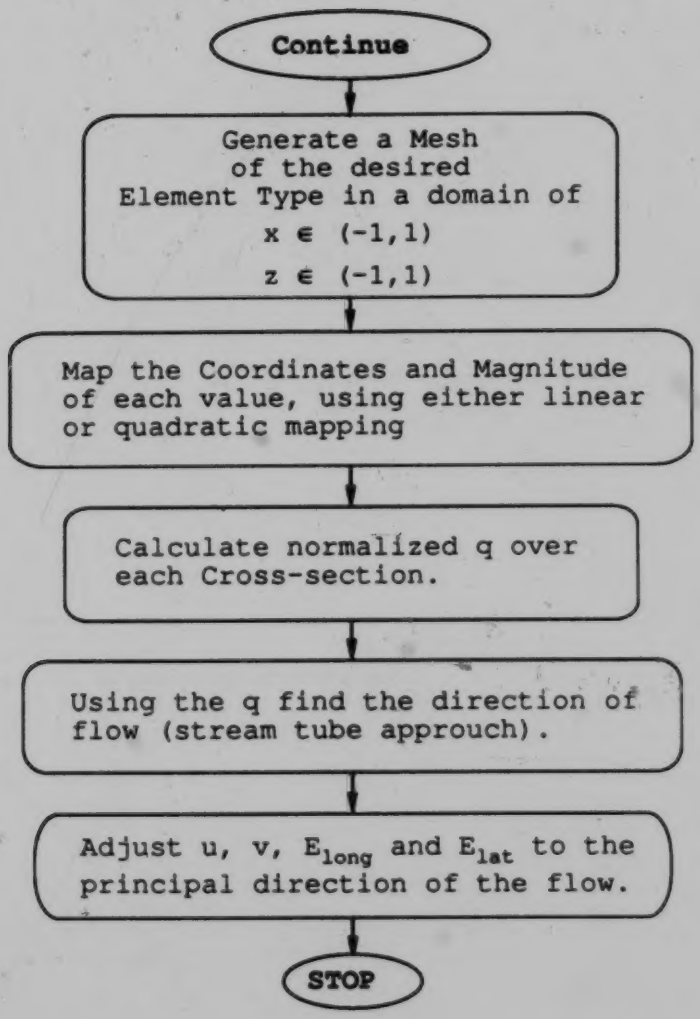

Continue
Generate a Mesh of the desired Element Type in a domain of
x ∈ (-1,1)
z ∈ (-1,1)
Map the Coordinates and Magnitude of each value, using either linear or quadratic mapping
Calculate normalized q over each Cross-section.
Using the q find the direction of flow (stream tube approuch).
Adjust u, v, E_{long} and E_{lat} to the principal direction of the flow.
STOP

Figure 3.6 Flow Chart 1 b, Generate a Mesh fitted to a River Reach

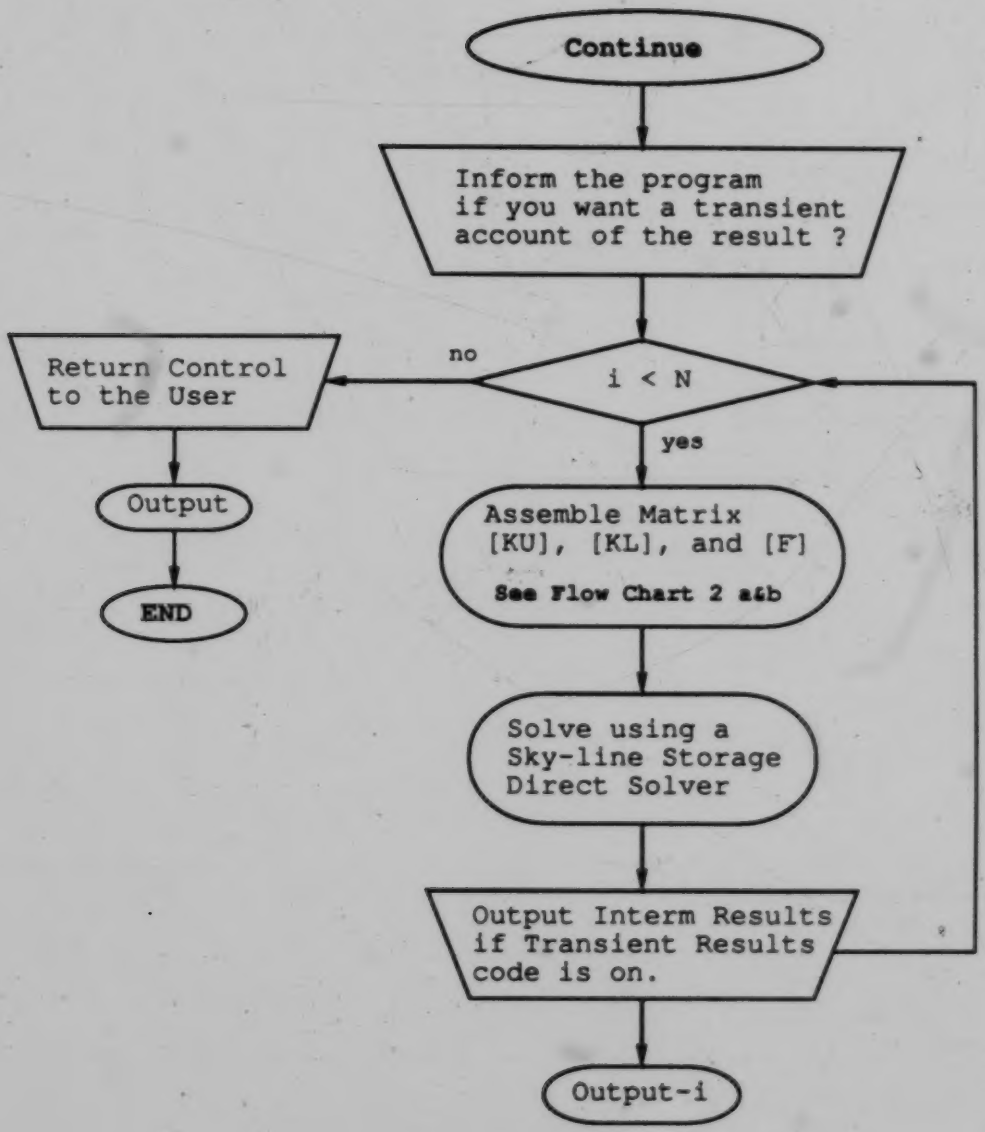

Continue
Inform the program if you want a transient account of the result ?
no
i < N
yes
Return Control to the User
Output
END
Assemble Matrix [KU], [KL], and [F]
See Flow Chart 2 a&b
Solve using a Sky-line Storage Direct Solver
Output Interm Results if Transient Results code is on.
Output-i

Figure 3.7 Flow Chart 1 c, The Programmed Solution Algorithm

Flow Chart 2a Assembling of Global Matrices Part I

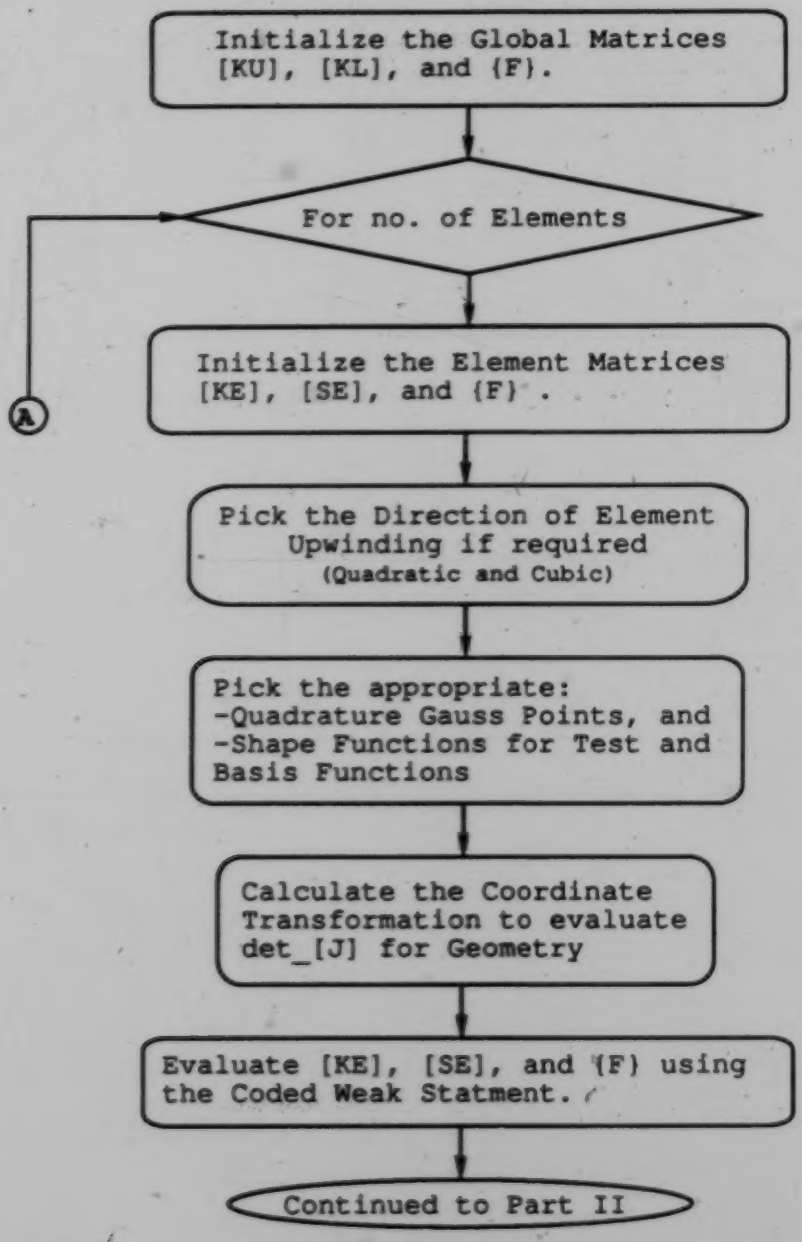

Initialize the Global Matrices [KU], [KL], and {F}.
For no. of Elements
Initialize the Element Matrices [KE], [SE], and {F} .
Pick the Direction of Element Upwinding if required (Quadratic and Cubic)
Pick the appropriate: -Quadrature Gauss Points, and -Shape Functions for Test and Basis Functions
Calculate the Coordinate Transformation to evaluate det_[J] for Geometry
Evaluate [KE], [SE], and {F} using the Coded Weak Statment.
Continued to Part II

Figure 3.8 Flow Chart 2 a, Assembling of Global Matrices, Part I

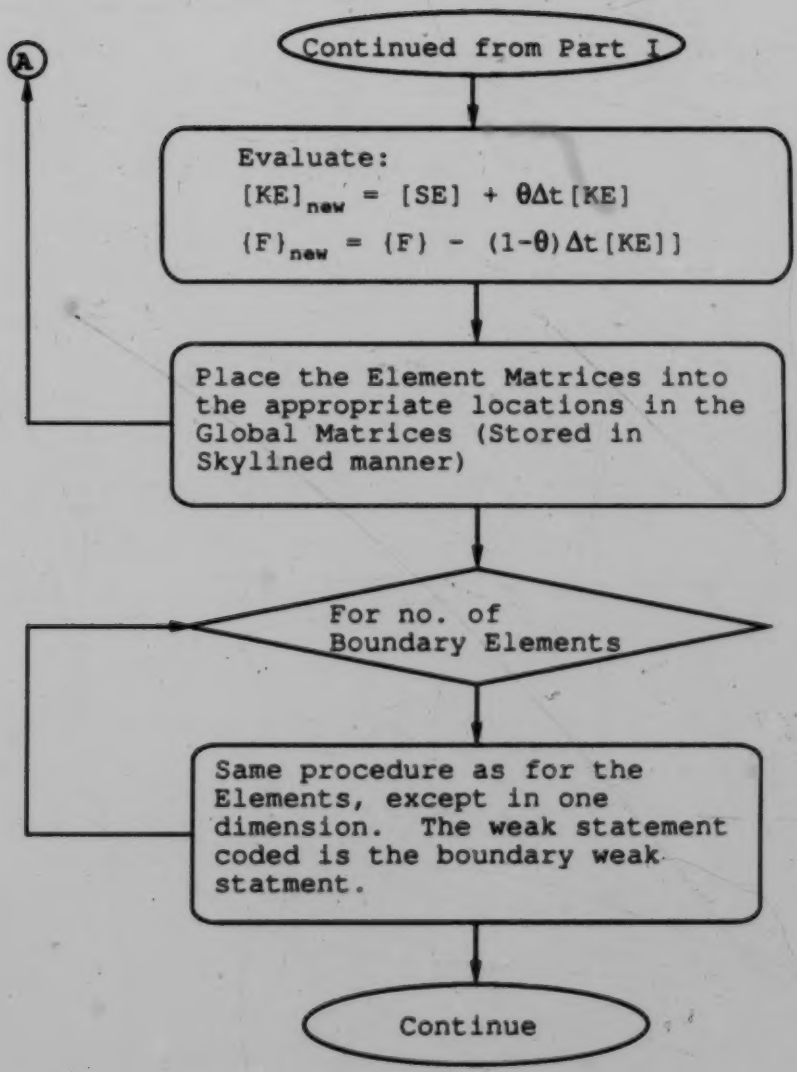

A
Continued from Part I
Evaluate:
$[KE]_{new} = [SE] + \theta\Delta t[KE]$
$\{F\}_{new} = \{F\} - (1-\theta)\Delta t[KE]]$
Place the Element Matrices into the appropriate locations in the Global Matrices (Stored in Skylined manner)
For no. of Boundary Elements
Same procedure as for the Elements, except in one dimension. The weak statement coded is the boundary weak statment.
Continue

Figure 3.9 Flow Chart 2 b, Assembling of Global Matrices, Part II

4. NUMERICAL TESTS

The performance of the upwinding basis functions introduced earlier has not yet been fully evaluated. The use of the these new elements was evaluated by performing a set of standard numerical tests. A comparison is presented in this chapter between the upwinding basis functions elements and other commonly used methods, in the analysis of pollutant discharges in rivers.

4.1 One Dimensional Comparison

The first comparison performed on these new elements was to test their behavior against the finite difference and finite element methods mentioned in Section 2.2.2, using an one dimensional test.

One of the most important characteristic desired of any numerical method built to solve the advection diffusion equation is its ability to accurately predict the behavior of advection dominated flows. A problem that will test this behavior is the solution of the unsteady advection Equation 2.28. All three methods formulated here were tested to determine their behavior in the solution of this equation.

4.1.1 LBG

The use of linear test and basis functions in the solution of Equation 2.28, results in the discrete equation shown below,

$$\left(\frac{1}{6} - \frac{\theta\rho}{2}\right)c_{i-1}^{n+1} + \frac{2}{3}c_i^{n+1} + \left(\frac{1}{6} + \frac{\theta\rho}{2}\right)c_{i+1}^{n+1}$$

$$= \left(\frac{1}{6} + \frac{(1-\theta)\rho}{2}\right)c_{i-1}^{n} + \frac{2}{3}c_i^{n} + \left(\frac{1}{6} - \frac{(1-\theta)\rho}{2}\right)c_{i+1}^{n} . \qquad (4.1)$$

The resulting Taylor series expansion of this discrete equation for $\theta = 0.5$ is

$$\frac{\partial C}{\partial t} + U\frac{\partial C}{\partial x} = O(\Delta x)^2 + O(\Delta t)^2. \qquad (4.2)$$

The Stone and Brian method discussed earlier in Section 2.2.2.2.1 is identical to this case when $\theta = 0.5$.

The dispersion and dissipation diagrams for this method are shown in Figure 4.1. It is apparent from the diagrams as well as from the Taylor Series expansion that this method shows no dissipation, but does show dispersion for the shorter wavelength perturbations. These small perturbations would persist in the solution, since there is no selective dissipation of them. The result of such an solution algorithm is the realization of oscillatory solutions.

This is shown in Figure 4.2, which is the solution for $\rho = 0.24$, of the advection of a Gaussian shaped pulse.

4.1.2 QUPG

The use of Quadratic basis functions and linear test functions results in the following discrete equation,

$$\left(\frac{-1}{24} + \frac{\theta\rho}{12}\right)c_{i-2}^{n+1} + \left(\frac{5}{24} - \frac{3\theta\rho}{4}\right)c_{i-1}^{n+1} + \left(\frac{17}{24} + \frac{\theta\rho}{4}\right)c_i^{n+1} + \left(\frac{1}{8} + \frac{5\theta\rho}{12}\right)c_{i+1}^{n+1}$$

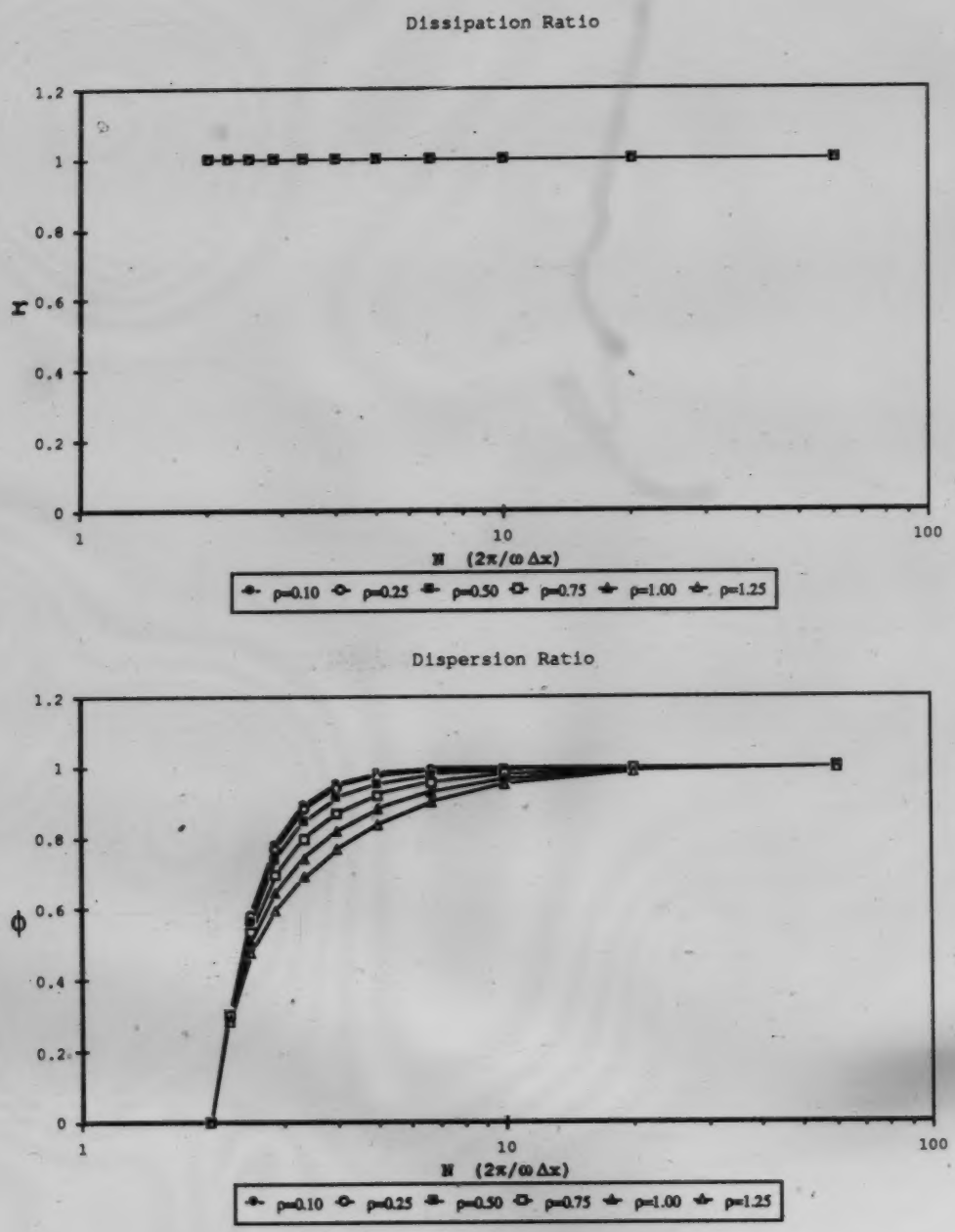


Figure 4.1 Dissipation and Dispersion for LBG

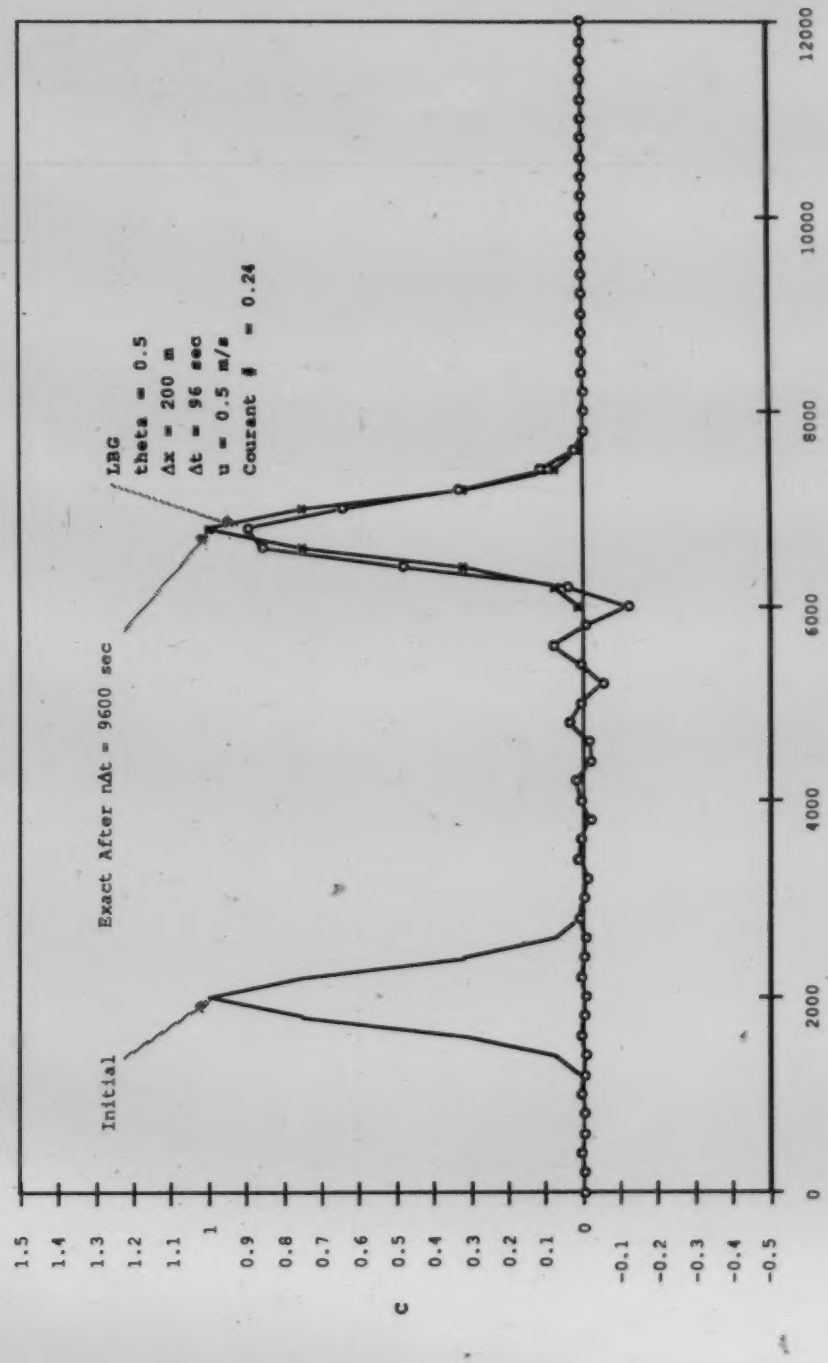


Figure 4.2 Advection of a Pulse using LBG

$$= \left(\frac{-1}{24} - \frac{(1-\theta)\rho}{12}\right)C_{i-2}^{n} + \left(\frac{5}{24} + \frac{3(1-\theta)\rho}{4}\right)C_{i-1}^{n} + \left(\frac{17}{24} - \frac{(1-\theta)\rho}{4}\right)C_{i}^{n}$$

$$+ \left(\frac{1}{8} - \frac{5(1-\theta)\rho}{12}\right)C_{i+1}^{n} \cdot \qquad (4.3)$$

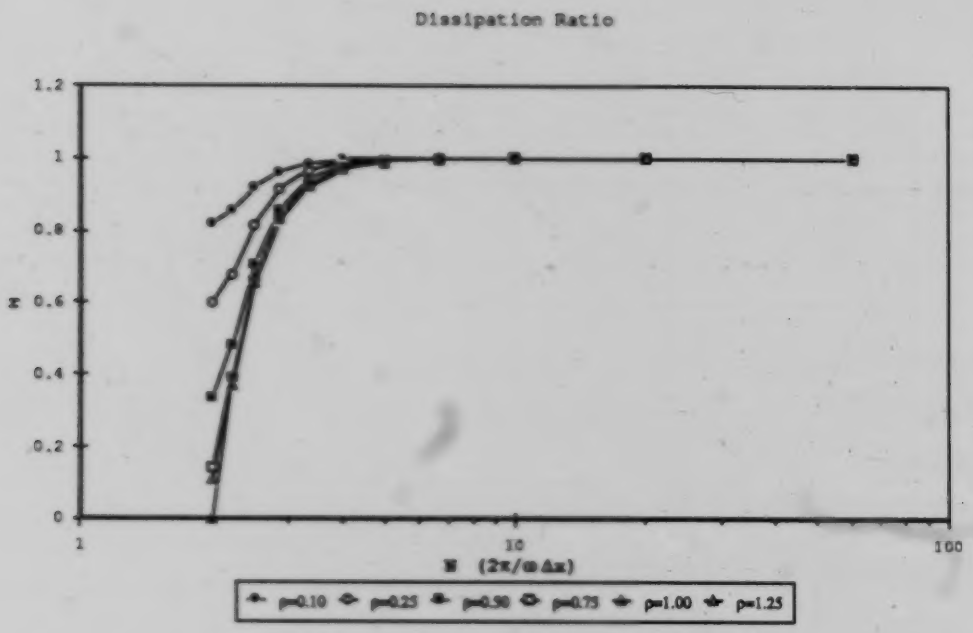
The modelled equation by this discrete equation is

$$\frac{\partial C}{\partial t} + U\frac{\partial C}{\partial x} + \frac{\Delta x^2 \partial^2}{12\,\partial x^2}\left(\frac{\partial C}{\partial t} + U\frac{\partial C}{\partial x}\right) + \frac{\Delta x^3 \partial^3}{24\,\partial x^3}\left(\frac{\partial C}{\partial t} + U\frac{\partial C}{\partial x}\right)$$

$$- \frac{\Delta x^4 \partial^4}{72\,\partial x^4}\left(\frac{\partial C}{\partial t} + U\frac{\partial C}{\partial x}\right) + \frac{\Delta x^4}{720} U\frac{\partial^5 C}{\partial x^5} + O(\Delta x^5) = 0, \qquad (4.4)$$

which can be seen to be fourth order accurate. The expansion of this equation would shows that no diffusion(that is the second derivative) is introduced by this equation. However there is some selective dissipation due to the higher order terms. The dispersion and dissipation diagrams for this method are shown in Figure 4.3. These diagrams show that though there is dispersion shown by the method, this dispersion is dissipated away. This implies that though the oscillation are present they disappear with time .

This is shown in the example solution of the advection of a Gaussian shape pulse at $\rho = 0.24$, illustrated in Figure 4.4. This solution shows the presence of a small wiggle trailing the pulse and the peak value is shown to be dissipated. However there is not the large amplified oscillations seen in the LBG method, as these are dissipated away as soon as they appear.



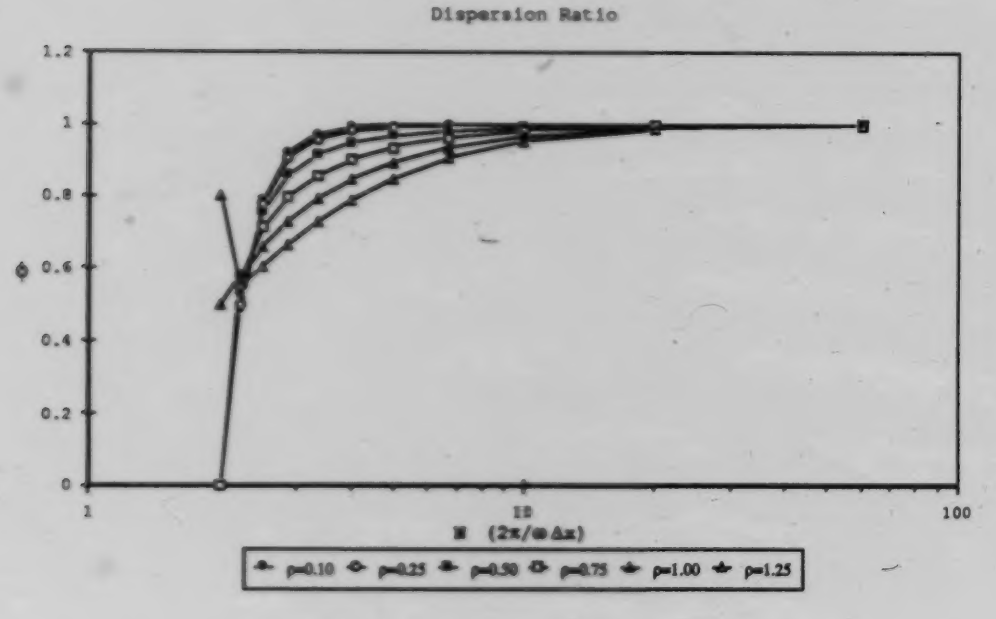


Figure 4.3 Dissipation and Dispersion for QUPG

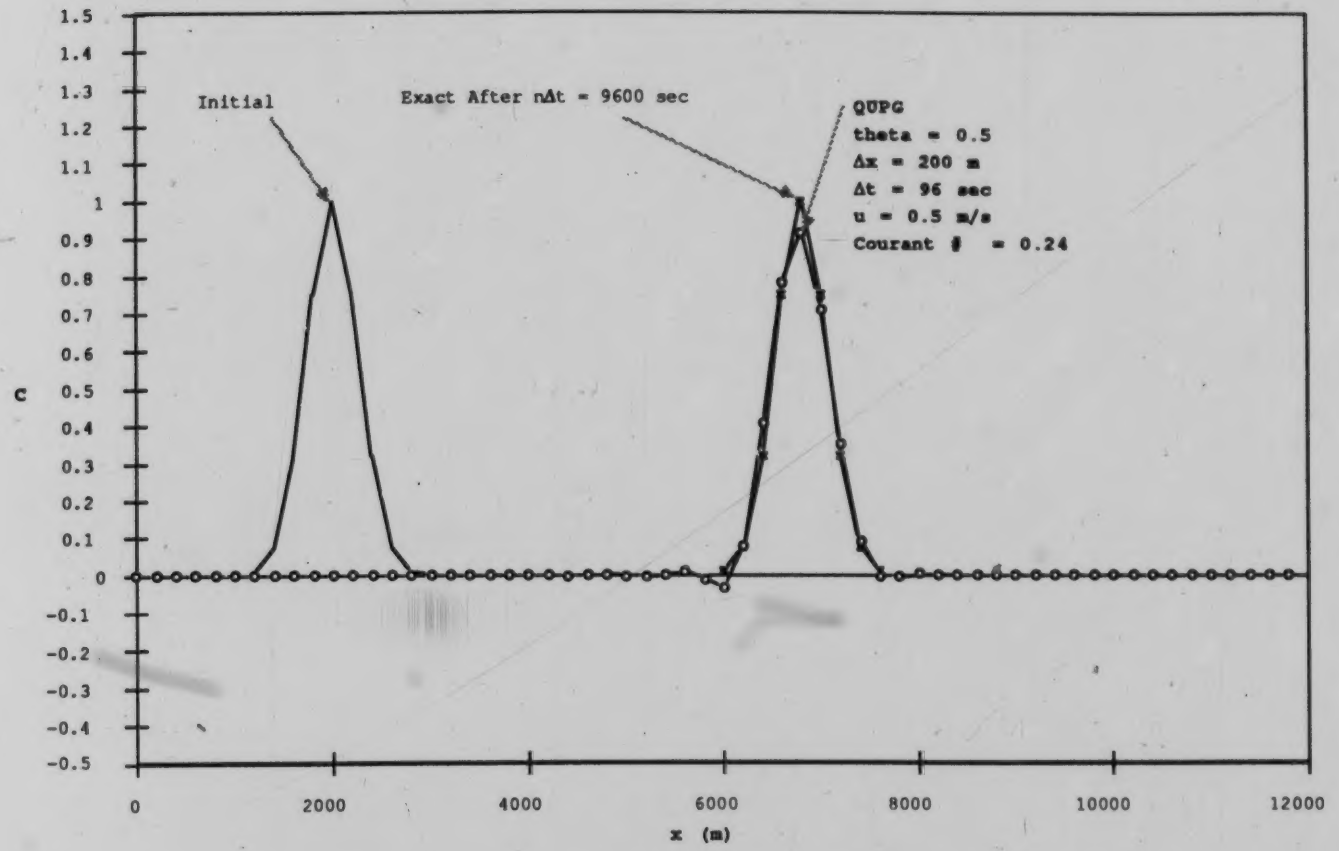


Figure 4.4 Advection of a Pulse using QUPG

4.1.3 CUPG

The discrete solution to Equation 2.10 at a particular node using Cubic basis functions and Linear test functions is represented by the equation,

$$\left(\frac{1}{45} - \frac{\theta\rho}{24}\right)C_{i-3}^{n+1} + \left(\frac{-4}{45} + \frac{\theta\rho}{4}\right)C_{i-2}^{n+1} + \left(\frac{13}{60} - \theta\rho\right)C_{i-1}^{n+1} + \left(\frac{67}{90} + \frac{5\theta\rho}{12}\right)C_{i}^{n+1}$$
$$+ \left(\frac{19}{180} + \frac{3\theta\rho}{8}\right)C_{i+1}^{n+1} = \left(\frac{1}{45} + \frac{(1-\theta)\rho}{24}\right)C_{i-3}^{n} + \left(\frac{-4}{45} - \frac{(1-\theta)\rho}{4}\right)C_{i-2}^{n}$$
$$+ \left(\frac{13}{60} + (1-\theta)\rho\right)C_{i-1}^{n} + \left(\frac{67}{90} - \frac{5(1-\theta)\rho}{12}\right)C_{i}^{n}$$
$$+ \left(\frac{19}{180} - \frac{3(1-\theta)\rho}{8}\right)C_{i+1}^{n} . \qquad (4.5)$$

The modelled equation by this discrete equation is

$$\frac{\partial C}{\partial t} + U\frac{\partial C}{\partial x} + \frac{\Delta x^2}{12}\frac{\partial^2}{\partial x^2}\left(\frac{\partial C}{\partial t} + U\frac{\partial C}{\partial x}\right) + \frac{7\Delta x^4}{240}\frac{\partial^4}{\partial x^4}\left(\frac{\partial C}{\partial t} + U\frac{\partial C}{\partial x}\right)$$
$$- \frac{\Delta x^5}{45}\frac{\partial^5}{\partial x^5}\left(\frac{\partial C}{\partial t} + U\frac{\partial C}{\partial x}\right) + \frac{\Delta x^5}{720}U\frac{\partial^5 C}{\partial x^5} + O(\Delta x^6) = 0, \qquad (4.6)$$

which can be seen to be fifth order accurate. Again the expansion of this equation shows no diffusion and little dispersion. The dispersion and dissipation diagrams for this method are shown in Figure 4.5, which shows smaller dispersion and dissipation rates at the lower wavelengths than those by QUPG. A peculiarity of CUPG is that there appears to be an optimum value for the Courant number which would give the almost no dispersion ($0.25 < \rho < 0.5$). This shows an ability of the elements to accurately produce, the exact solution at such a Courant number. One can also

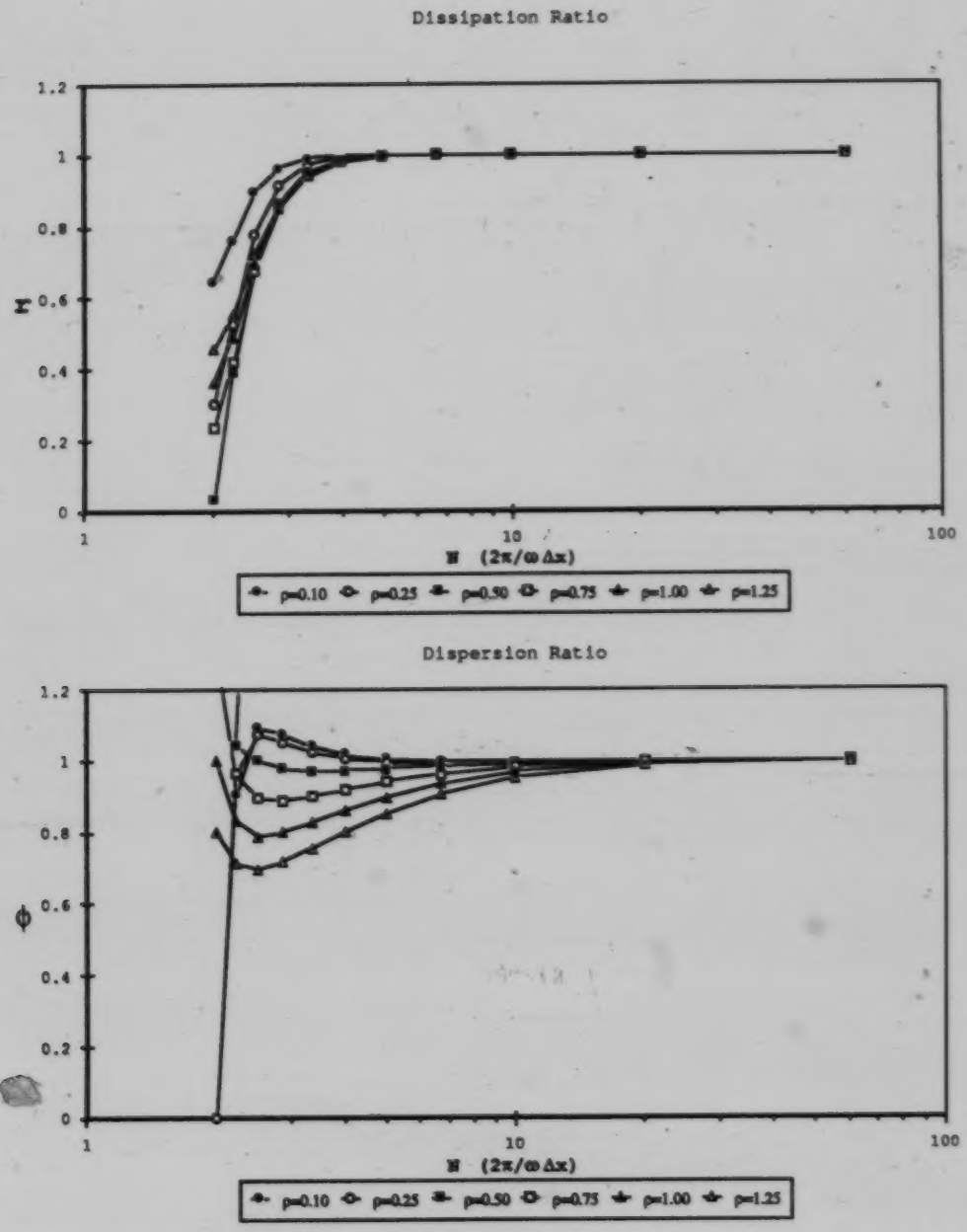


Figure 4.5 Dissipation and Dispersion for CUPG

observe that at Courant numbers less than 1.0, the small wavelength perturbations will be advected at phase speeds greater than exact. This implies the presence of wiggles ahead of the main pulse.

A plot of the solution for the advection of a Gaussian shaped pulse is shown in Figure 4.6 for $\rho = 0.24$. This illustration shows a peak value higher than the QUPG and the presence of a wiggle trailing the pulse, which is again smaller than that for QUPG. It also show the presence of a wiggle in front of the pulse due to the dispersion mentioned earlier.

An overall comparison of the these three and the four finite difference methods is shown in Figure 4.7. The indicators used to judge the behavior of the method are the L2 discrete norm., the maximum value, and the minimum value, each indicating the accuracy the dissipation and the dispersion respectively. The discrete L2 is defined as

$$L2' = \frac{\sum (C_i - C_{exact})^2}{\text{Total Mass}}, \qquad (4.7)$$

which has a value of zero for an exact solution. The maximum value for the exact solution should be 1.0 and the minimum should be zero.

The semi-log bar chart of the L2' shows clearly that the best finite difference scheme at the Courant number of 0.24 is the Holly-Preissmann method (Sauvaget, 1985). While the

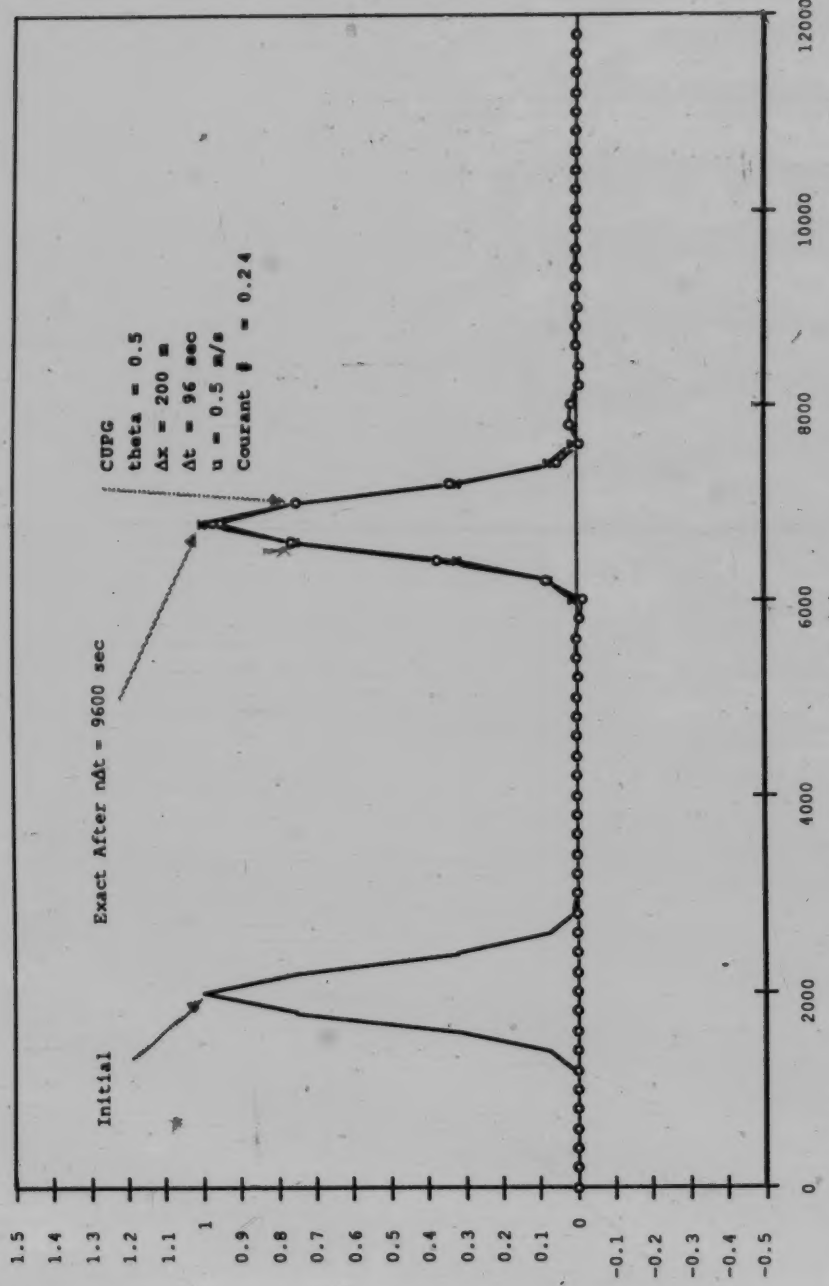


Figure 4.6 Advection of a Pulse using CUPG

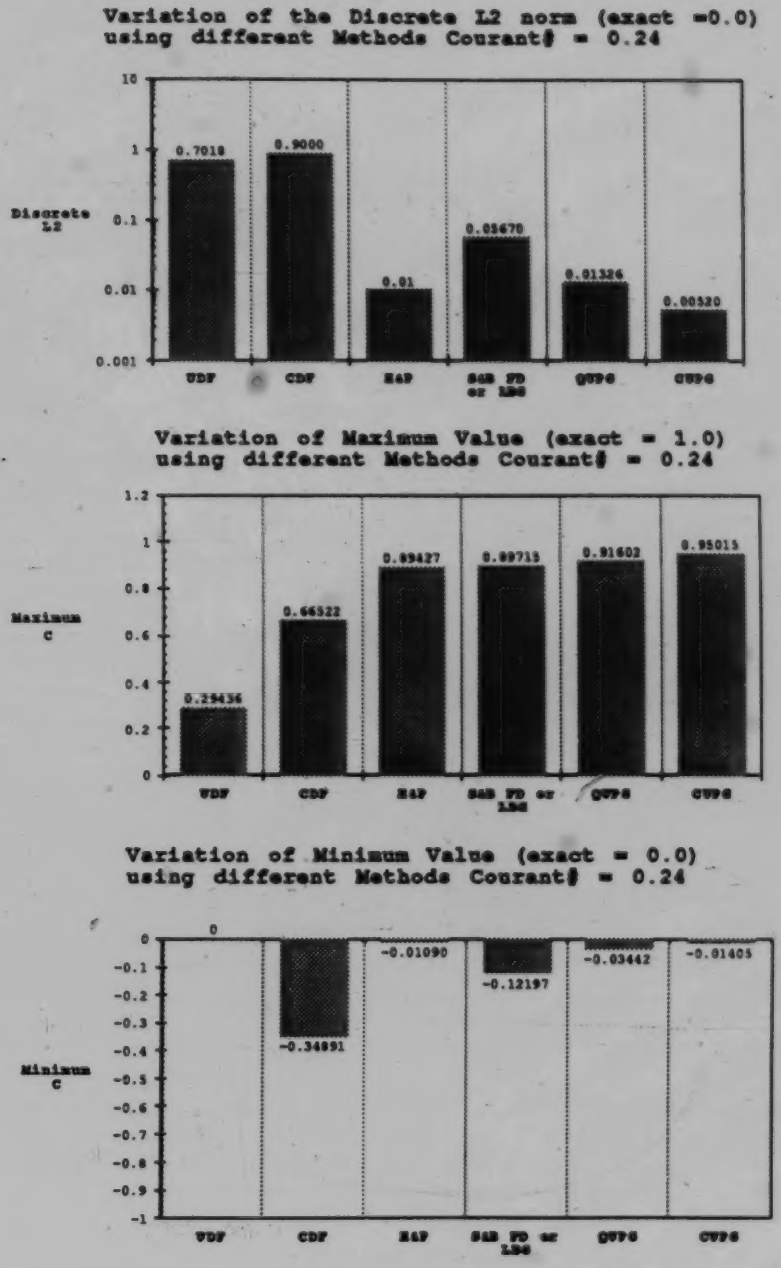


Figure 4.7 Comparison of One D tests

best finite element method is CUPG. The most accurate all of the methods tested is CUPG.

A similar conclusion can be derived for the charts of the maximum values. While for the minimum values, are the upwind finite difference method is the best, due to its artificial diffusion.

It should be noted that though a Courant number of 0.24 is representative of the type of value used in a general case, it does not represent the optimum choice for any particular method. However to model any general problem, which could conceivably have a range of different Courant numbers, the test can't be restricted to special Courant numbers. As a conclusion it is evident from the this analysis that the finite element method gives by far the best results in the one dimensional case.

4.2 Two Dimensional Tests

The previous analyses showed that the finite element method gives the best results in the one dimensional cases and since the scope of this thesis is to apply the finite element method to river problems, the remainder of the thesis will restrict itself to the evaluation of the two dimensional finite element method. Two two dimensional numerical test were performed to judge the advantages gained from new element types. The first was to look at the behavior of the

new elements in simple advection equation in two dimensions. The second consisted of looking at the steady state behavior of a plume in skewed meshes.

4.2.1 Two Dimensional Advection

To look at the distortion resulting from a given choice of element types in a simple two dimensional advection, the equation

$$\frac{\partial C}{\partial t} + U \frac{\partial C}{\partial x} + V \frac{\partial C}{\partial y} = 0 \qquad (4.8)$$

was solved. The mesh size, velocity field and initial condition used for the test are as follows,

$\Delta x = 200$ m; $\Delta y = 200$ m; $x \in (-3400, 3400)$; $y \in (-3400, 3400)$;

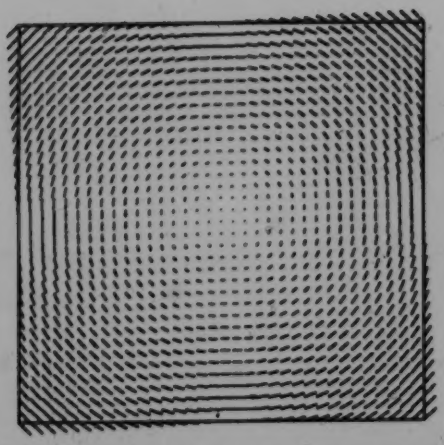
$U = \omega x$; $V = \omega y$; $\omega = \frac{2\pi}{3000}$;

$$C_o(x,y) = e^{\left(-\frac{(x-x_o)^2+(y-y_o)^2}{2\,\sigma_o^2}\right)}$$

where $x_o = 0.0$, $y_o = -1800.0$, and $\sigma_o = 264.0$. The boundary conditions used for this case were all fixed value of 0.0 on all the boundaries.

The velocity vectors and the contour plots, as well as other salient information, for the solutions using the three different element types for median Courant numbers of 1.9 and 0.19, are shown in Figure 4.8 and 4.9 respectively.

Table 4.1 summaries the indicators used to judge the behavior of the the methods for both the Courant numbers.

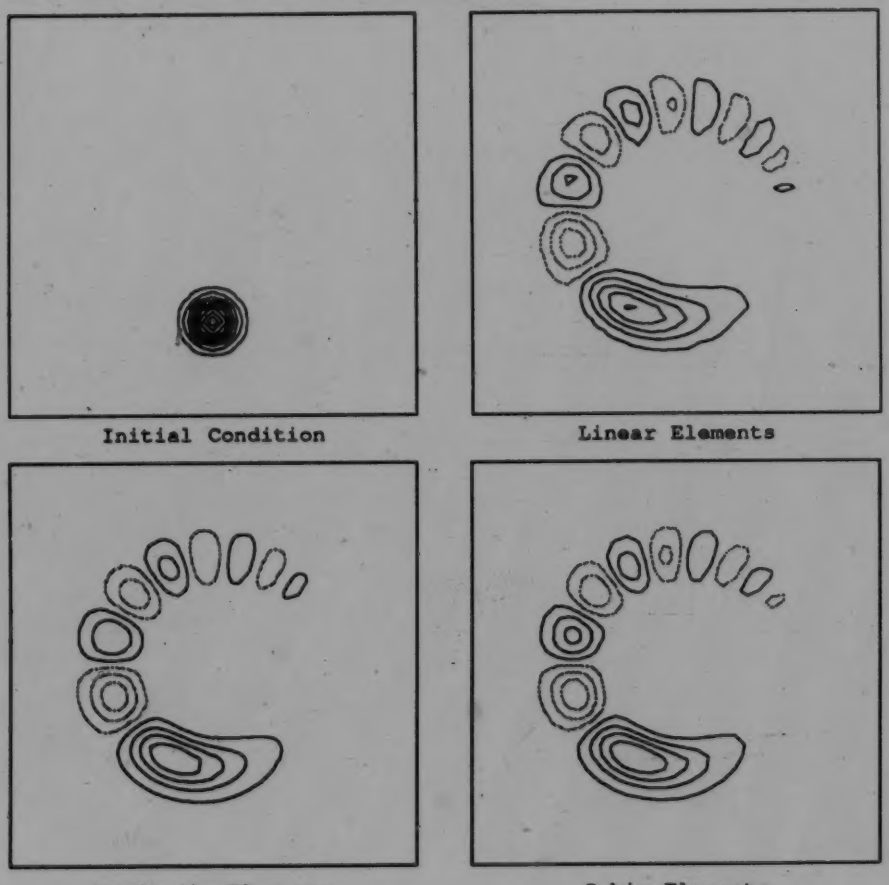


$\Delta x = \Delta y = 200$ m
$\Delta t = 100$ sec
Median Courant no. = 1.9
Theta = 0.5
$u(x) = 3000x/2\pi$
$v(y) = 3000y/2\pi$
total t = 3000 seconds

Contour Particulars:
Solid lines indicate C > 0.0
Dashed lines indicate C < 0.0.
Contour Interval = 0.1
Max Contour:

Initial :	1.0
Linear :	0.45
Quadratic :	0.35
Cubic :	0.35

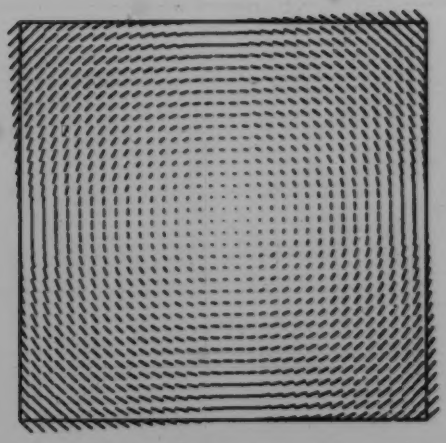
Velocity Vectors (flow Counter Clockwise)



Initial Condition | Linear Elements

Quadratic Elements | Cubic Elements

Figure 4.8 Circular Advection of a Gauss Hill at a Courant # ~ 1.9



$\Delta x = \Delta y = 200$ m
$\Delta t = 10$ sec
Median Courant no. = 0.19
Theta = 0.5
$u(x) = 3000x/2\pi$
$v(y) = 3000y/2\pi$
total t = 3000 seconds

Contour Particulars:
Solid lines indicate C > 0.0
Dashed lines indicate C < 0.0.
Contour Interval = 0.1
Max Contour:

Initial :	1.0
Linear :	0.75
Quadratic :	0.75
Cubic :	0.85

Velocity Vectors (flow Counter Clockwise)

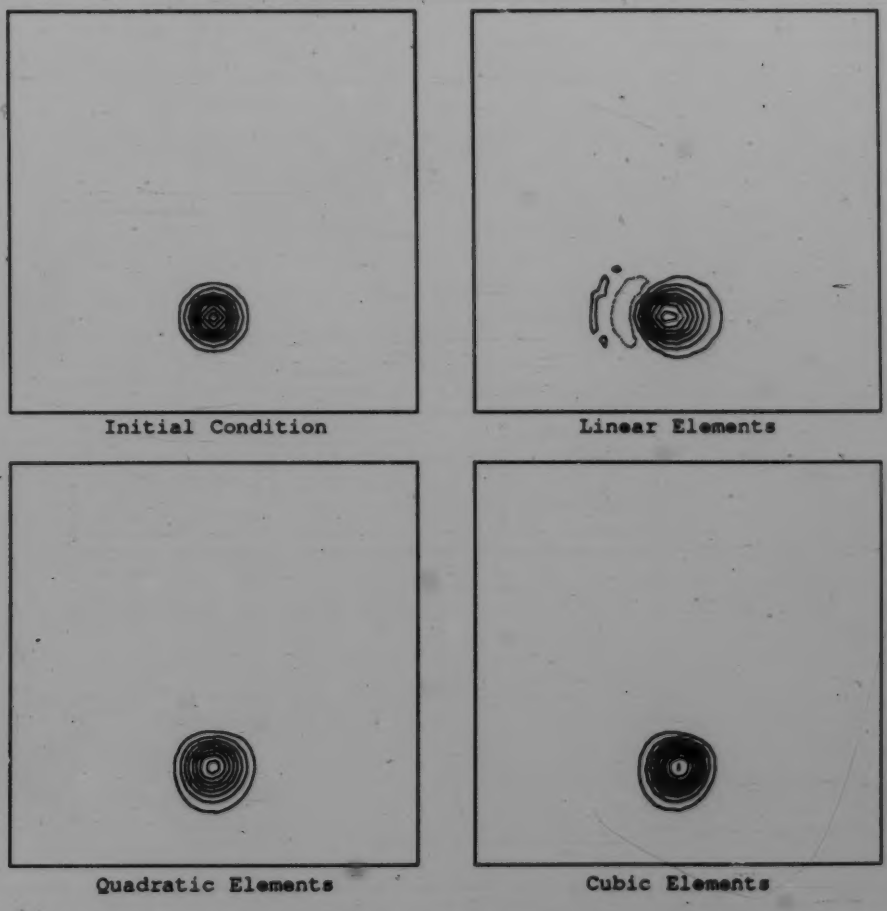


Figure 4.9 Circular Advection of a Gauss Hill at a Courant # ~ 0.19

C=1.9		LINEAR	QUADRATIC	CUBIC *
L-2 error Norm		0.0010900	0.0009489	0.0011117
Error in Peak		0.4633100	0.4254800	0.4517400
Error in Max neg.		-0.3127300	-0.3121500	-0.3227900
Error in Position				
of Peak	R	0.9938080	0.9162457	0.9493337
	Theta	0.9262082	0.9610104	0.9428999
Discrete L-2 Norm		5.64079E-06	4.76991E-06	5.53459E-06
Max Difference		0.6937200	0.6175700	0.6968200

C=0.19		LINEAR	QUADRATIC	CUBIC *
L-2 error Norm		0.0003307	0.0002116	0.0001494
Error in Peak		0.8229000	0.7799600	0.8526500
Error in Max neg.		-0.1207300	-0.0306500	-0.0364700
Error in Position				
of Peak	R	1.0061539	1.0000000	1.0000000
	Theta	0.9823884	1.0000000	1.0000000
Discrete L-2 Norm		1.76957E-06	1.05947E-06	7.38396E-07
Max Difference		0.2805892	0.2200400	0.1473500

* indicates the use of an iterative solution technique as discussed in 3.1.7

Table 4.1 Accuracy Measurements for the Advection of a Gauss Hill

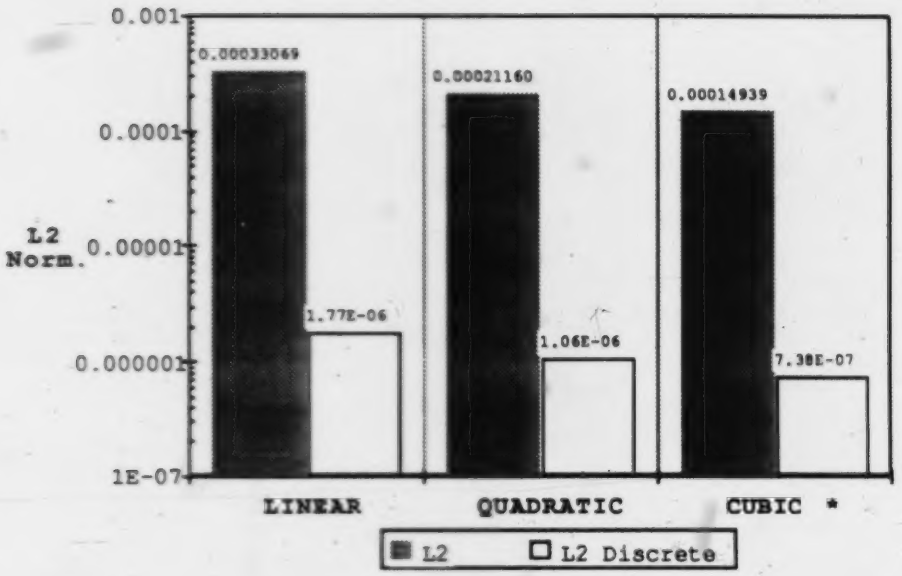
The indicators used to make the judgement are, the L2 error Norm defined by the expression

$$L2 = \frac{\int_{\Omega} (C_{num} - C_{exact})^2 \, d\Omega}{\int_{\Omega} C_{exact} \, d\Omega} \; ; \qquad (4.9)$$

error in peak (Max), which should have an exact value of 1.0; error in minimum (Min), which should have an exact value of 0.0; error in the position as indicated by the value of the peak's radius and rotation compared to the exact solution; the discrete L2 norm as defined by Equation 4.7; and finally inf error, which is the maximum difference between the exact and the numerical solution.

It is apparent from the contour plots for the test at a Courant number of 1.9, that all three element types give a similar type of distorted solutions. This implies that at such a high Courant number the use of upwinding basis functions is of no added advantage. The same conclusions can be derived from poor performance shown by the bar charts of the error measures for the three methods shown in Figure 4.10.

At a median Courant number of 0.19, the solutions look to be very reasonable. LBG shows its wiggles behind the pulse, while QUPG shows a nice smooth solution. Also evident



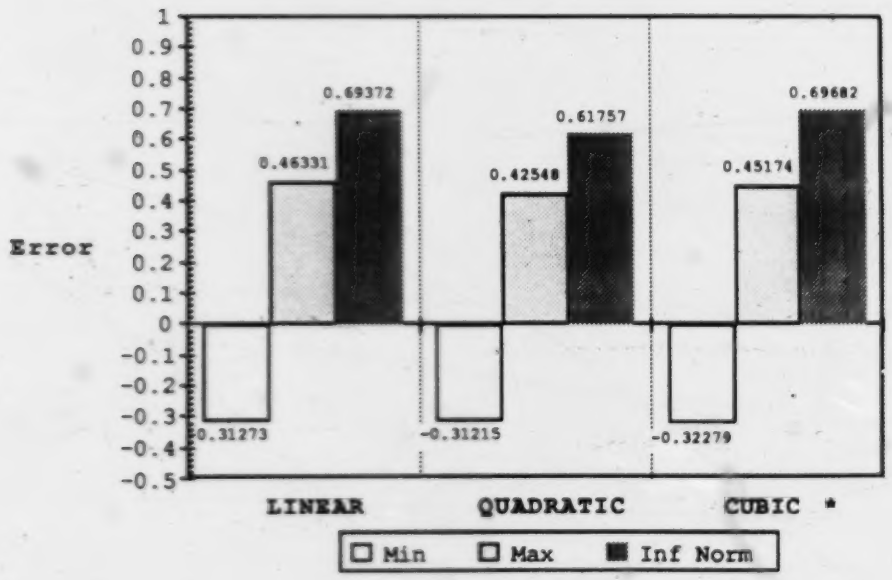


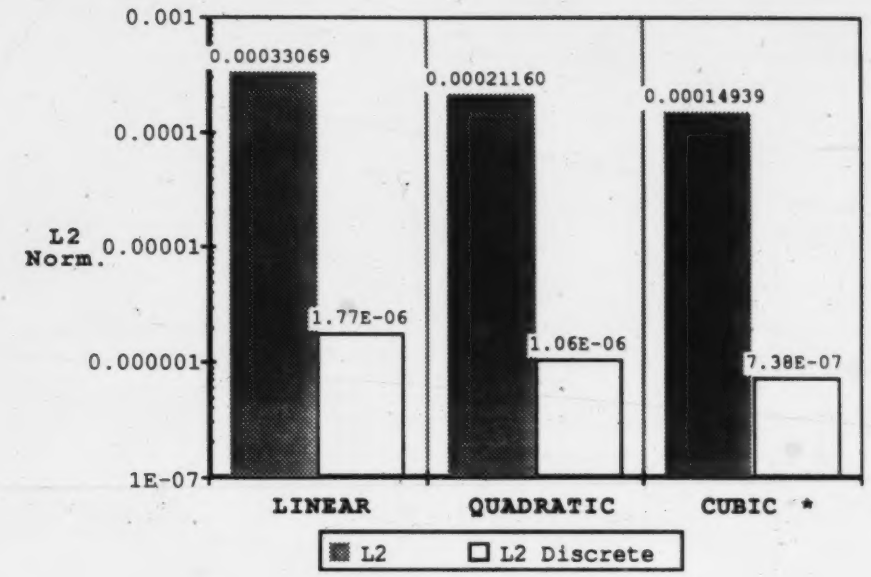
Figure 4.10 Error measures for 2 D advection Courant # 1.9

from the contour plots and the bar charts of the error measures in Figure 4.11, is that QUPG shows slightly more dissipation than CUPG, which is in agreement with results found for the one dimensional solution. The L2 norm seems to decrease in intervals of a third with each increase in order accuracy of the elements. That is it is reduced by a third from LBG to QUPG and a third form QUPG and CUPG. The inf norm reduces in a similar manner, expect it seems to decrease in 25% increments. There is an improvement in the size of wiggle (Min.) from LBG to QUPG (about 75%), however there only a slight improvement from QUPG to CUPG. The peak is preserved the best by CUPG. Generally the best performance is again given by CUPG.

4.2.2 Steady Plume in Skewed Meshes

One of the most important factor that indicate the applicability of a numerical method in practical situations is its ability to solve accurately in meshes that are skewed to the flow direction. A method that has this ability can accurately predict solutions in flows that are not aligned with the mesh such as circulations.

To compare the behavior of the new elements in a skewed mesh situation a steady problem was used. The problem consisted of a plume discharge into a body of water which had a constant velocity in a given direction.



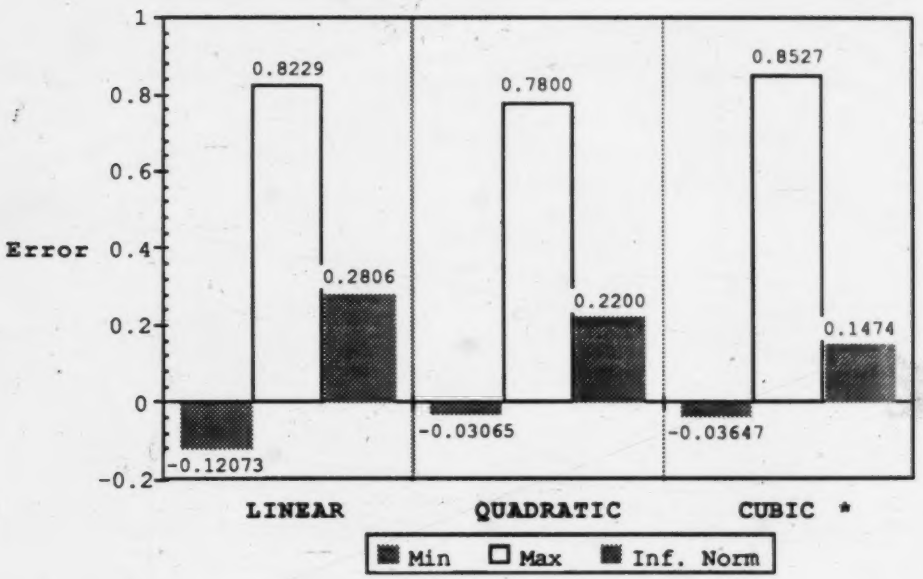


Figure 4.11 Error measures for 2 D advection Courant # 0.19

Two different cases were looked at to study this behavior. One in which the plume was solved for in four different directions skewed to the mesh, using the same method. While second case consisted of the use of the three element types to solve for a plume in a given direction of flow to the mesh.

4.2.2.1 Case 1

To infer the behavior to a natural channel situation, an 'infinitely' wide channel of constant depth of 2.0 m, slope of 0.0005, $C_\star$ of 25.0 and velocity of 2.475 m/s was used. Using a grid size of 1.0 m and both the longitudinal and the lateral diffusion given by $0.2hU_\star \simeq 0.0391$, results in cell Peclet ($P_e \equiv \frac{U\Delta x}{E}$) number of 63.3. The flow was skewed to the mesh at angles of 0°, 30°, 45° an 60°.

The resulting concentration contours from using LBG are shown in Figure 4.12. The best solution is realized when the skew angle is zero and if the angle is changed, a similar solution should be obtained. However, since the LBG method prefers to have the flow in a favorable direction (ie 0° or 45°), there are oscillation produced. A summary of the salient features in presented in Table 4.2. Of interest in this case is the high negative concentrations for the 30° and 60° skew angle, which are in complete error. The test shows that LBG has a preference in the direction of the flow with respect to the mesh.

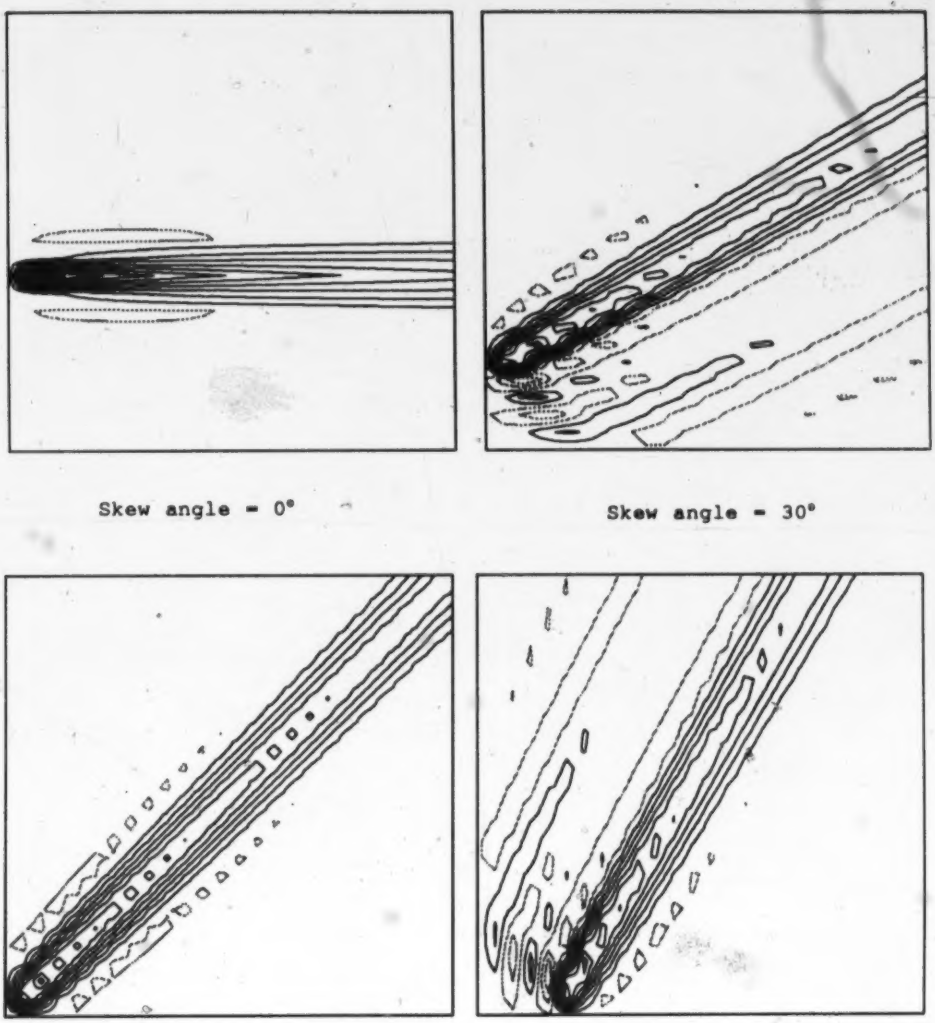


Figure 4.12 Solutions for different Skew angle Using LBG

Case 1
Solutions using Linear Elements
at Different angles of Skew

Source = 1.0 g/s
U = 2.475 m/s
E_long = E_lat = 0.2hU* = 0.0391
Slope = 0.0005
C* = 25.0
depth = 2.0 m
Δx = Δy = 1.0 m
Cell Peclect no. = 63.3
All extremas are normalized
by the Concentration at the source

Angle of Skew	*Ratio of Cmax to C@source*	*Ratio of Cmin to C@source*	*Concentration at the Source*
0°	1.2609	-0.0323	0.3841
30°	1.0000	-0.4247	0.3986
45°	1.0492	-0.0436	0.4061
60°	1.0000	-0.4247	0.3986

Case 2
Solutions for 60° Skewness using different elements

Source = 5.0 g/s
U = 2.475 m/s
E_long = E_lat = 0.2hU* = 0.0391
Slope = 0.0005
C* = 25.0
depth = 2.0 m
Δx = Δy = 100.0 m
Cell Peclect no. = 6330
All extremas are normalized
by the Concentration at the source

Angle of Skew	*Ratio of Cmax to C@source*	*Ratio of Cmin to C@source*	*Concentration at the Source*
Linear	1.0787	-0.5218	204.5
Quadratic	1.1624	-0.3004	169.3
Cubic	1.2670	-0.1934	156.9

Table 4.2 Summary of Skew test, Case 1 and Case 2

4.2.2.2 Case 2

The comparison of LBG to QUPG and CUPG in their ability to handle skewed mesh, was performed by looking at the solution given by each element type for a skew angle of 60°. The P_e was also hiked up so as to increase the difficulty each method has to resolve the solution. This was accomplished by increasing Δx and Δy to 100 m, which results in a 100 fold increase in the cell Peclet number to 6330. The strength of the source was also increased to 5.0 g/s so as to present a reasonable plume.

The contour plots of the resulting solutions are shown with the related details in Figure 4.13. Table 2b summaries the error measures used to judge the solutions. The maximum and minimum concentrations reach by each method indicate that LBG performs very poorly, as expected. However these indicator look considerable better for the case of CUPG. Thus the new elements seem to handle skewed meshes with far better success that the standard linear ones.

This conclusion can also be seen from Figure 4.14, which shows the concentration profile through the source in the y direction. From this plot it is evident that QUPG and CUPG both show one wiggle behind the core of the plume, while the rest of the domain is free from the large oscillations, which are so dominant in LBG.

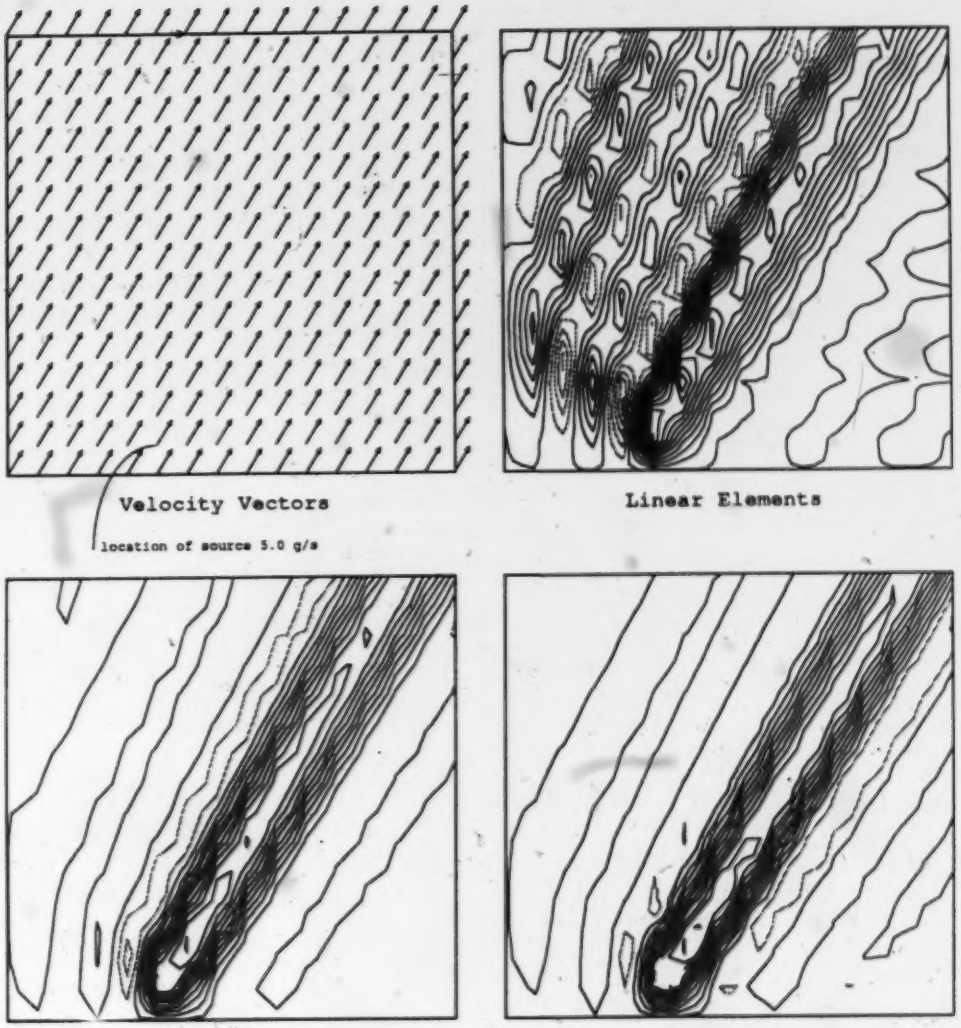


Figure 4.13 Solutions for different Methods for 60° skew angle, Pe = 6330

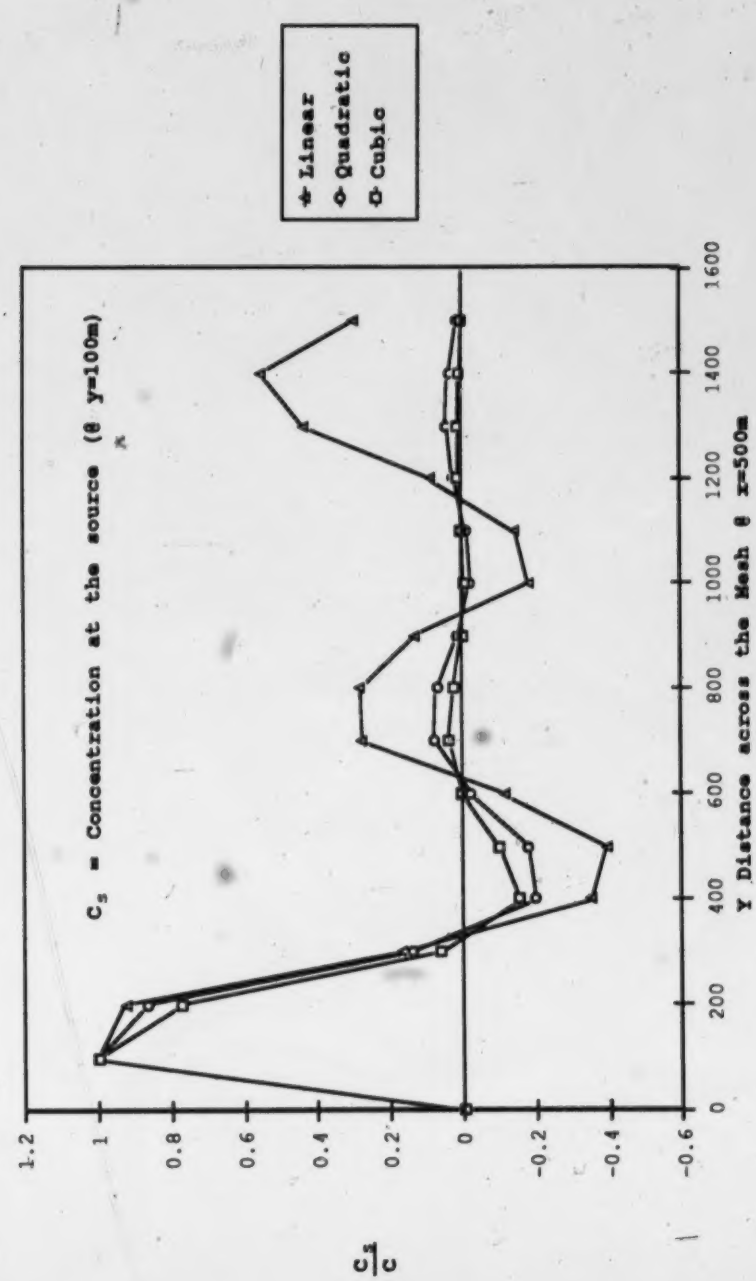


Figure 4.14 Concentration Profile across the Solutions for Skew test.

4.3 Conclusion of Numerical Tests

The tests performed here on the new elements indicate that their performance is better than the standard LBG method. However it should be noted that this increase in accuracy means an increase in the computational effort required. The use of QUPG and CUPG increases the computing time and the storage requirements for a given problem.

Figure 4.15 show that in the solution of a 20 by 20 mesh using banded storage, the storage required increases by a factor of 2 and 3 for the QUPG and CUPG elements respectively. The computational time on a Macintosh II machine increases by 3 and 6 times for the two element types. The use of a particular type of element must be weighted between the accuracy required and the computational resources available for a given situation.

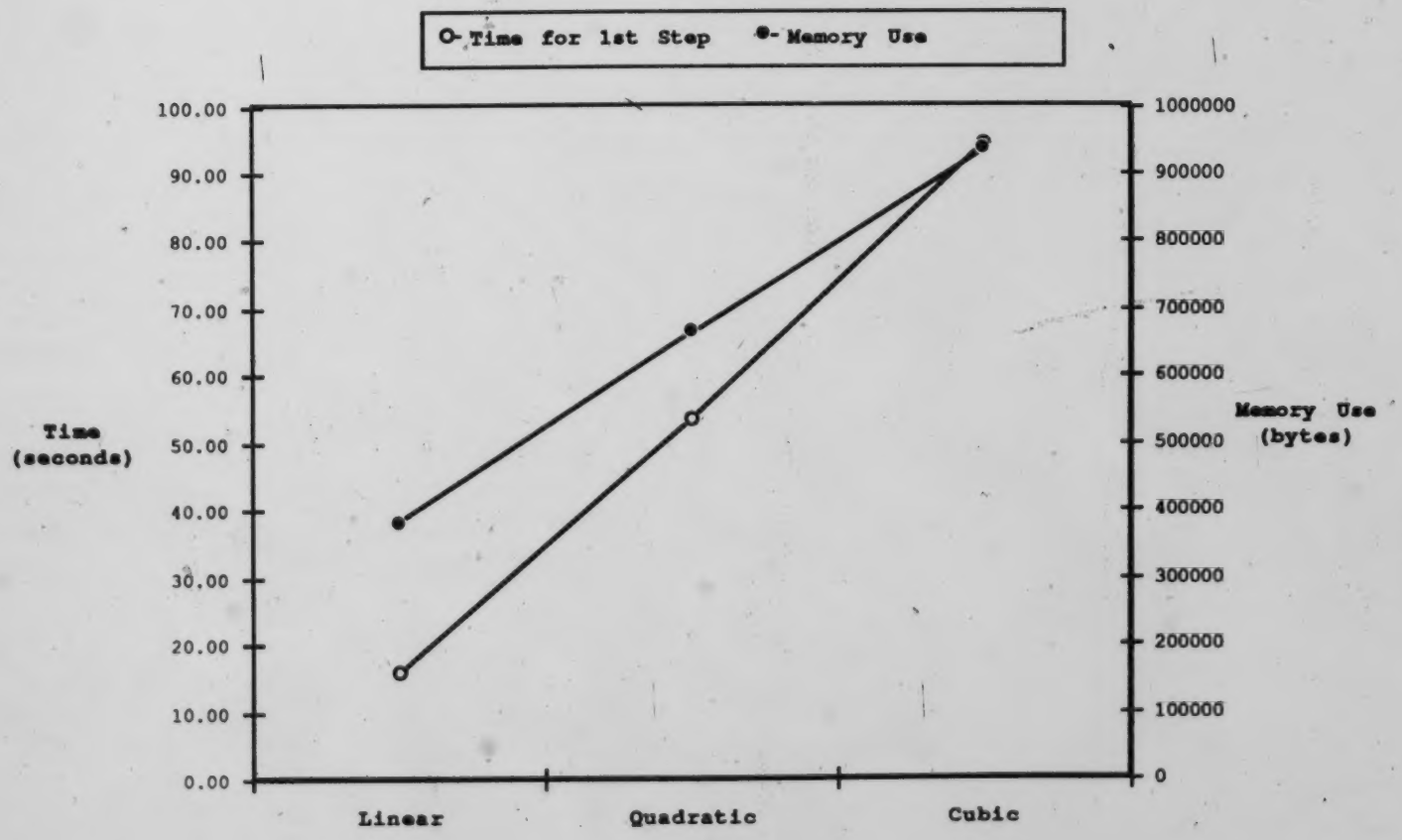


Figure 4.15 Computational effort required for the three methods

5. APPLICATIONS

Ideal and practical problems were solved using the numerical model to understand and reinforce the physical aspects of effluent discharges into natural channels. The idealized investigation consisted of studying the relationship between plume characteristics and modelling techniques used to access the variation of eddy diffusivity in a river channel. The second set of studies looks at the application of the numerical model to a practical river channel situations.

5.1 Variation of Eddy Diffusivity.

One of the hardest parameters to determine for a plume discharged into a river is its diffusive behavior. Generally it is believed that the simplifying assumption made, such as the use of constant eddy diffusivity is sufficiently accurate for practical analysis. However by making this assumption what important two dimensional characteristics are we ignoring ? To answer such a question two study problems were solved. One was to look at the discharge of a plume in the center of a channel having a cross-section shaped like a triangle. The other consisted of a side discharge into a trapezoidal shaped channel.

5.1.1 Center Discharges

The first investigation consisted of the discharge of a plume in the center of a channel with a triangular shaped cross-section. A triangular shape was chosen so as to exaggerate the effects, and to infer the behavior to a cross-section similar shape in the bends of rivers. The channel width was chosen to be 50 m and a maximum depth of 1.0 m was set on one side, resulting in an aspect ratio of 50. The length of the channel was chosen to be 8 km. The source of the plume was discharged into the river centered around the 27.5 m. The mesh used to model this plume was 20 elements long by 30 elements wide which set $\Delta x = 400$ m and $\Delta z = 1.67$ m. The slope of the channel was set to 0.0005 m/m. The resulting cell Peclet numbers were $\frac{40000}{h}$ and $\frac{278}{h}$ in the x and z direction respectively, where h is the depth.

The mesh was solved using the LBG formulation. There did not exist a need to use higher ordered elements since the discretization was well graded. As well the cell Peclet number were relatively small, particularly in the z direction, where the large changes are taking place. In addition the mesh was aligned to one of the preferred directions, so that the solution would not introduce any errors due to mesh being skewed to the direction of flow.

The two solution were performed, one with the use of E_{zz} as a function of the depth, and another with E_{zz} being

constant across the channel width. The E_{zz} as a function of the depth was given by the equation

$$E_{zz} = 0.15\ h\ U_* \qquad (5.1)$$

and the average value given by the expression

$$\overline{E_{zz}} = \frac{1}{Na} \sum_{i=1}^{Na} (0.15hU_*)_i \ , \qquad (5.2)$$

where Na is number of nodes across the channel(31). In both cases the value of E_{xx} was set by the expression,

$$E_{xx} = 0.25\ h\ U_* \ . \qquad (5.3)$$

Figure 5.1 show the distributions of E_{xx}, E_{zz}, and U across the channel resulting from these conditions.

The resulting contours from the two solutions as well as a plot of the mesh and velocity vectors is shown in Figure 5.2. Figure 5.3 shows the spreading characteristics of the plume half width long the channel. From the contours plots as well as the concentration profiles, it is evident that the use of an average eddy diffusivity across the channel alters the spreading rate of the plume. The plot shows that in the case where an average diffusivity is used, the spreading is larger in the shallow portion of the channel, while in the deep portion the spreading is smaller. This is due to the distribution of the diffusivity across the channel. In the shallow portion $\overline{E_{zz}}$ is larger than E_{zz} and thus the spreading is larger. While in the deep portion $\overline{E_{zz}}$ is smaller than E_{zz} and therefore the plume spreads less. For comparison, the spreading rate of a point source given by the integral

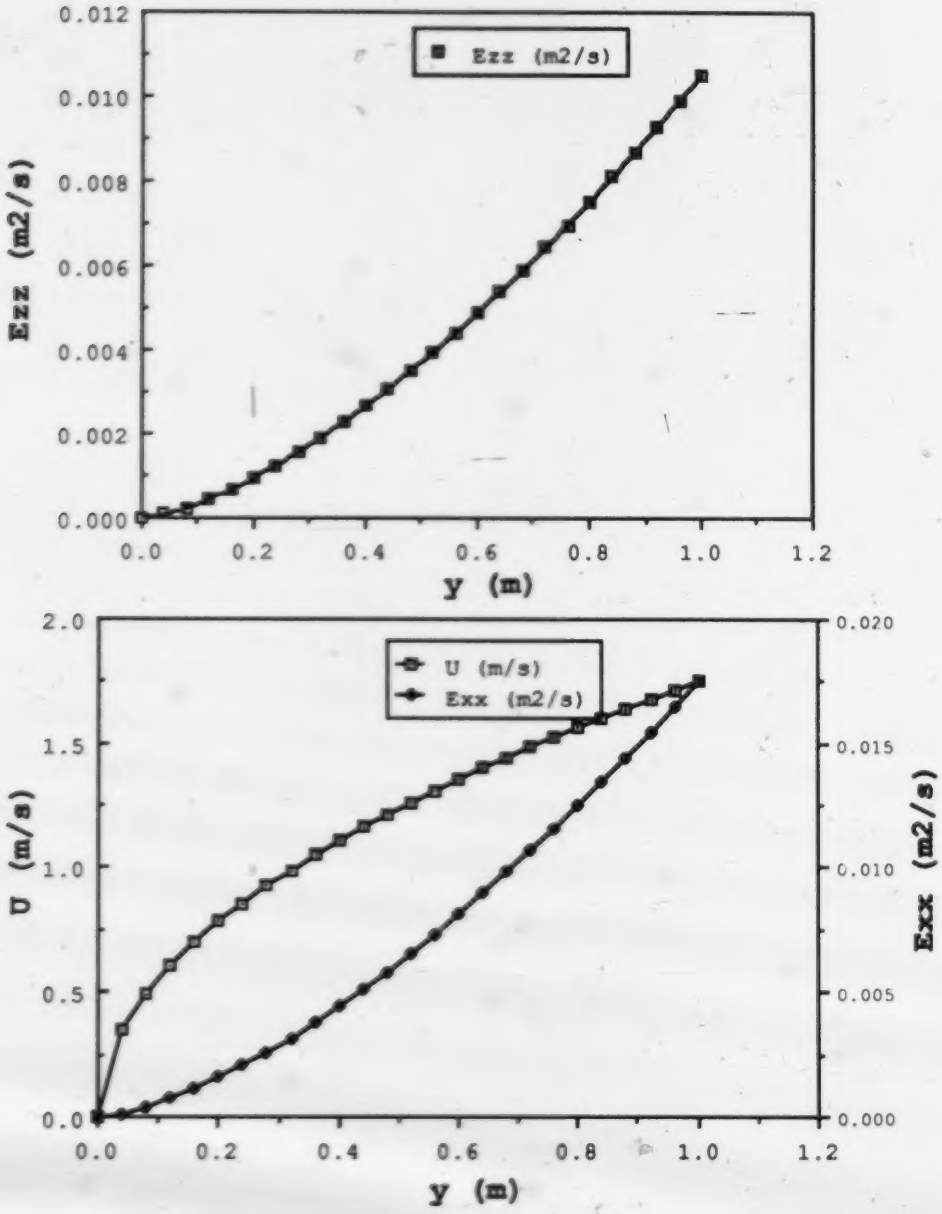
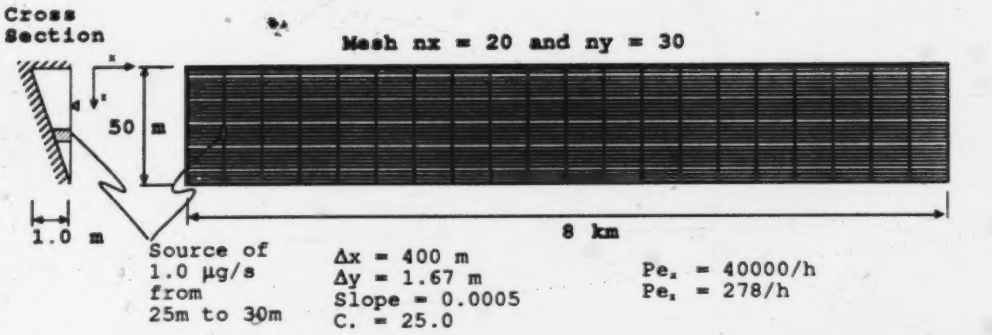
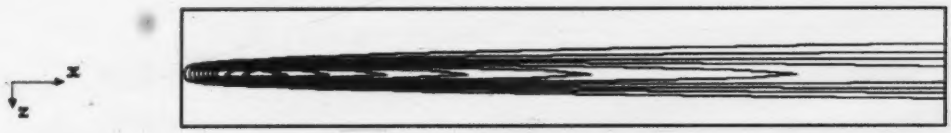


Figure 5.1 Variation of Exx, Ezz and U

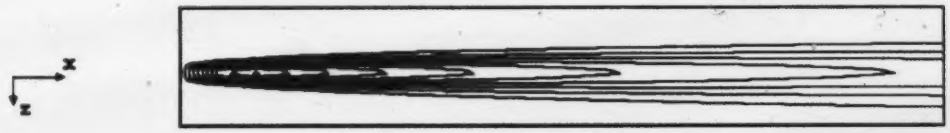


Solution with a variable E_{zz}



E_{xx} = 0.25hu. E_{zz} = 0.15hu.

Solution with an average E_{zz}



E_{xx} = 0.25hu $\overline{E_{zz}}$ = 1/n(Σ0.15hu)

Figure 5.2 Results for Triangular Channel

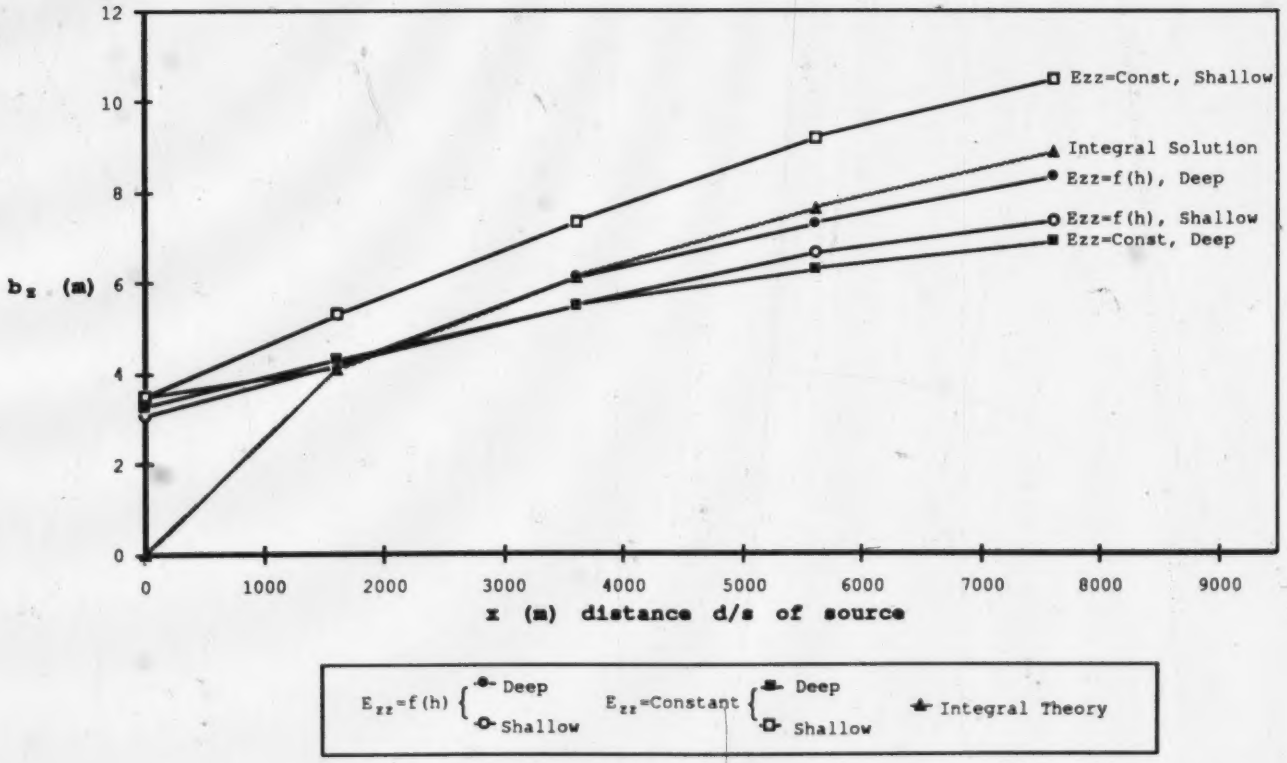


Figure 5.3 Plume Spreading Rates in a Triangular Cross-Section Channel

solution is also included in the plot(Rajaratnam, 1970). This solution fits nicely into the middle of the profiles, indicating its integral nature.

The longitudinal diffusion characteristics of the plume is shown in Figure 5.4, which shows a plot of the center line concentration along the channel. From this plot it is evident that the use of $\overline{E_{zz}}$ exaggerates the spreading rate, as indicated by the lower concentrations than those for the case of when E_{zz} was used.

In general, it can be concluded that for a plume situated in the center of the channel the use of $\overline{E_{zz}}$ across the channel will result in smearing of any anomalies that may exist due the the variation of the depth.

5.1.2 Side Discharges

The second situation studied was that of a side plume discharge into a channel with cross-section shaped like a trapezoid. The parameters used were the same as for the triangular channel. However the depth was kept at a value of 1.0 m for distance of 36.0 m from 3.0 m onward. The slope of the trapezoidal side was chosen as 3 horizontal to 1 vertical. The discharge was modelled by introducing a source of strength 1.0 µg/s over the sloping portion of the channel at a distance 160 m from the beginning of the left-hand side

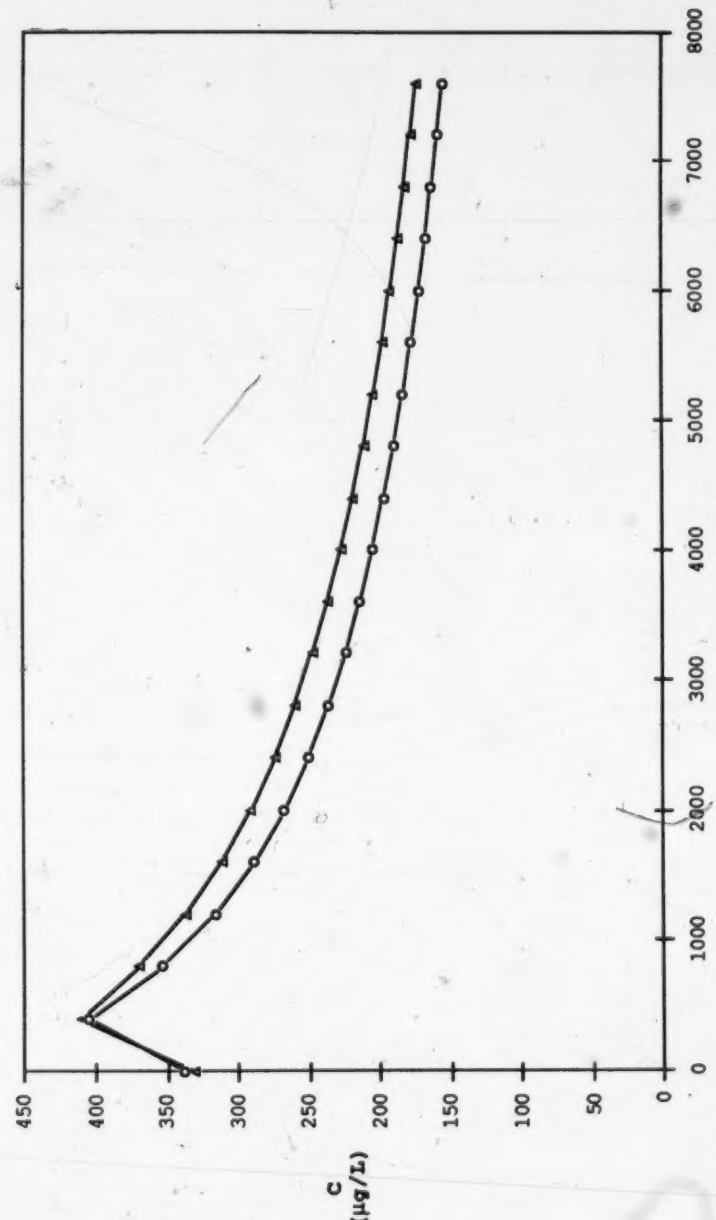


Figure 5.4 Concentration Profile in the Center of Plume Along the Channel

of the mesh. An illustration of the mesh, velocity vectors, and a summary of the salient features is shown in Figure 5.5.

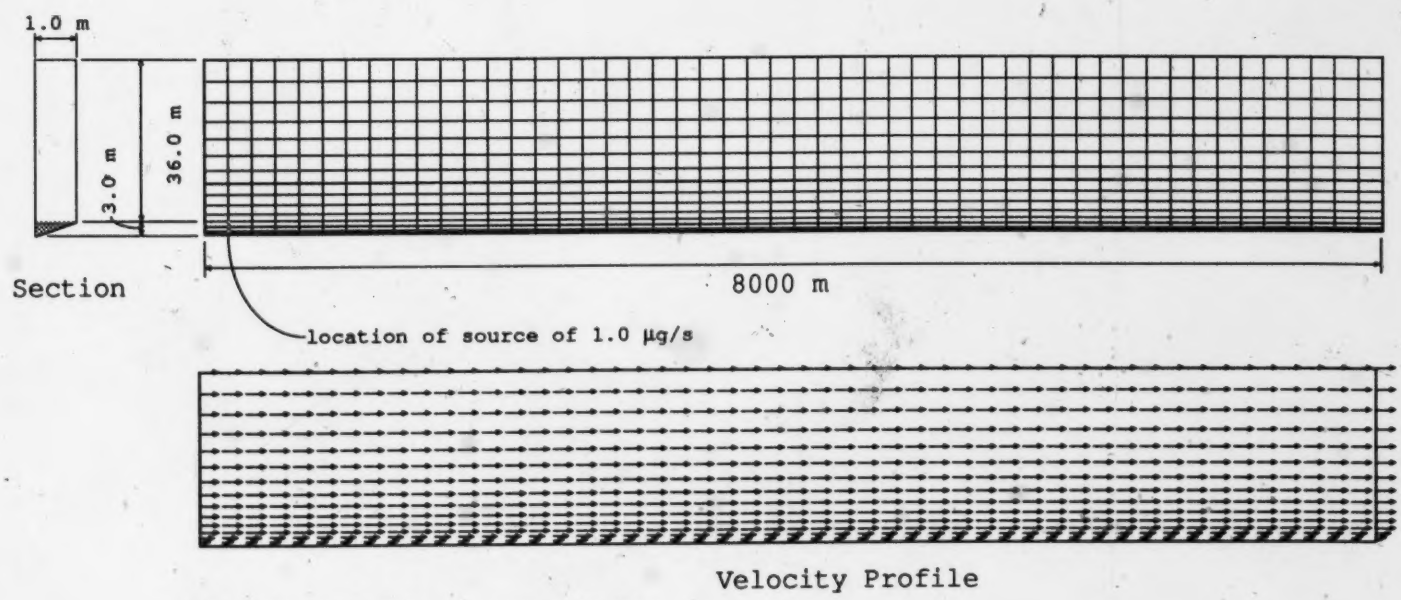
The problem was solved for the four different scenarios shown below,

(a)	E_{zz} = f(h)	E_{xx} = f(h)
(b)	E_{zz} = Constant	E_{xx} = f(h)
(c)	E_{zz} = Constant	E_{xx} = Constant
(d)	E_{zz} = Constant	E_{xx} = 0.0

Case (a) and (b) were used to observe the behavior due to the variation of E_{zz}, while (b), (c) and (d) were performed to observe the behavior due to E_{xx}.

The contour plots for each of the above case are shown in Figure 5.6. In addition a close-up of the contours for cases (a) and (b) are shown in Figure 5.7. A plot of the half-width(b_z) is also shown in Figure 5.8, and a plot of the concentration profile along the bank is shown in Figure 5.9.

These plots clearly show the difference between using a constant E_{zz} and a variable E_{zz}. As was found for the case of the center discharge earlier, the spreading rate of the plume is increased when an average value is used for E_{zz}. As a consequence the concentration on the bank is a great deal higher for case (a), than for case (b). The close-up of the contour plot shows another visual difference between the two cases. The use of a constant E_{zz} results in numerical result where the concentration contours are orthogonal to the



Δx min.= 0.24 m; max.= 5.0 m
Δy = 160.0 m
E_{xx} = 0.25 h U_*
E_{zz} = 0.15 h U_*
U_* = 0.0700 m/s
S = 0.0005
h = 1.0 m
C_* = 25.0

Figure 5.5 Mesh and Velocity Vectors

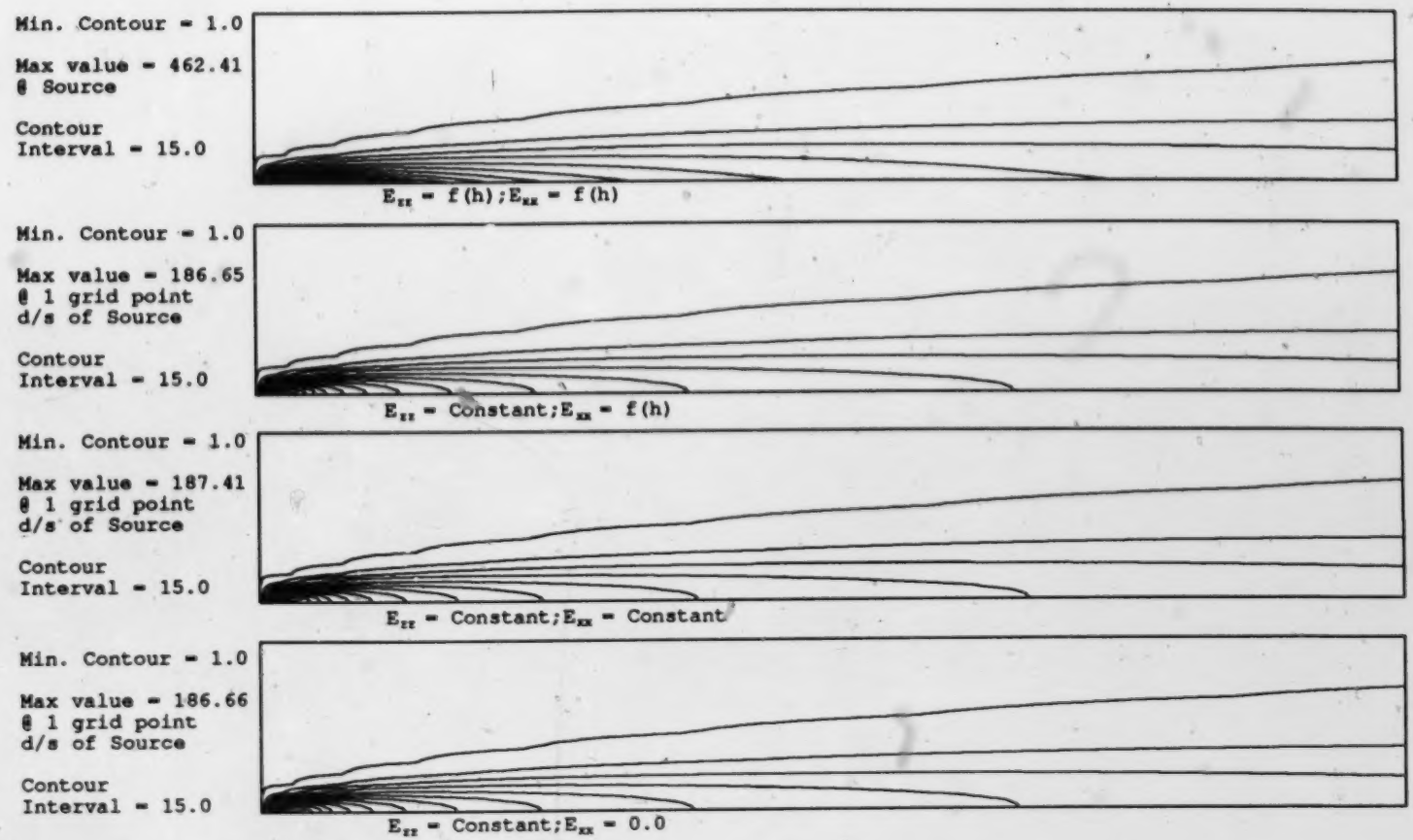


Figure 5.6 Contour Plots for Trapazoidal Shape Channel

Min. Contour = 1.0

Max value = 462.41
@ Source

Contour
Interval = 15.0

$E_{zz} = f(h); E_{xx} = f(h)$

Min. Contour = 1.0

Max value = 186.65
@ 1 grid point
d/s of Source

Contour
Interval = 15.0

$E_{zz} = \text{Constant}; E_{xx} = f(h)$

Figure 5.7 Close-up of the Contours

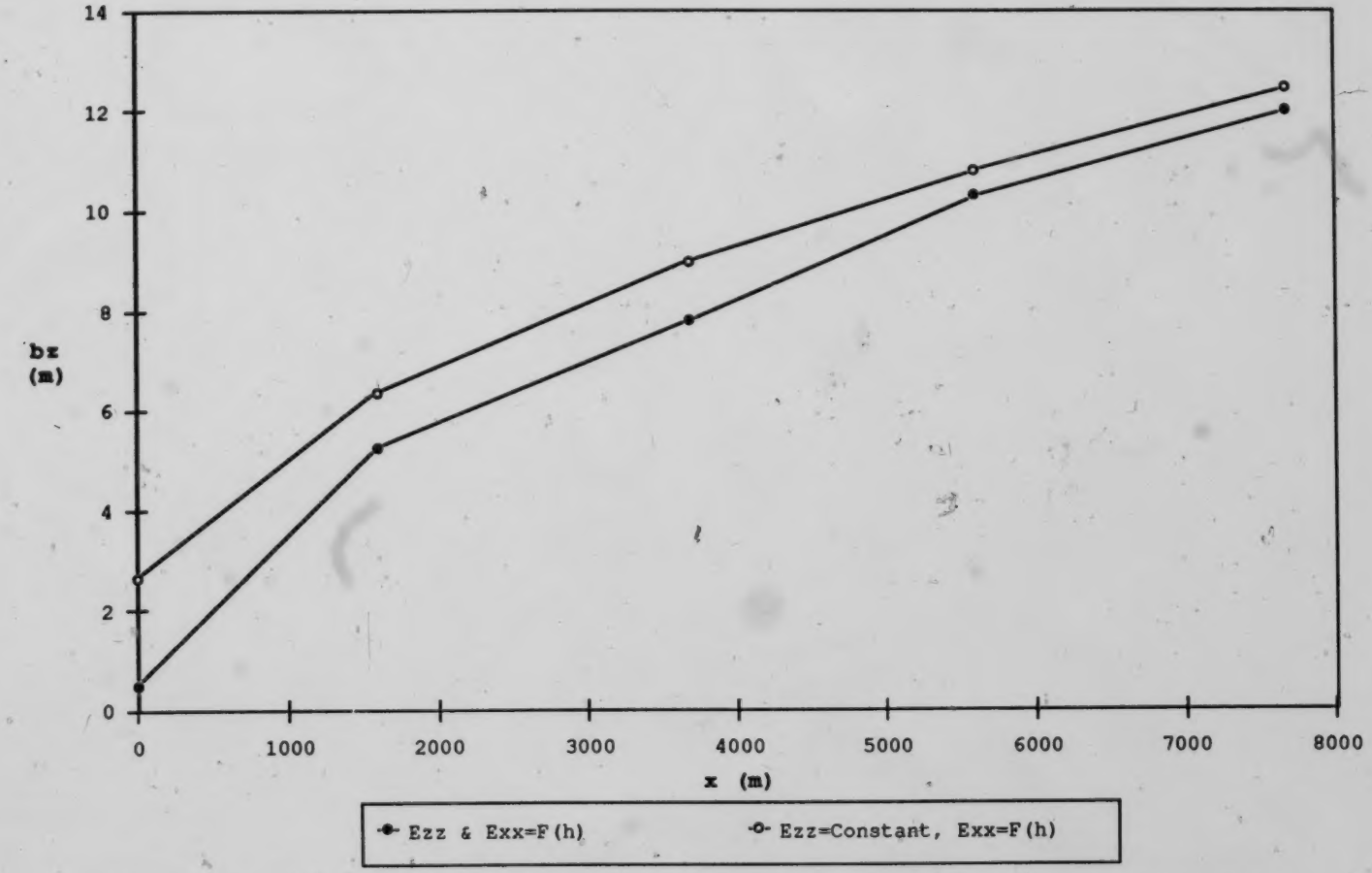


Figure 5.8 Variation of bz, the plume half width along the channel

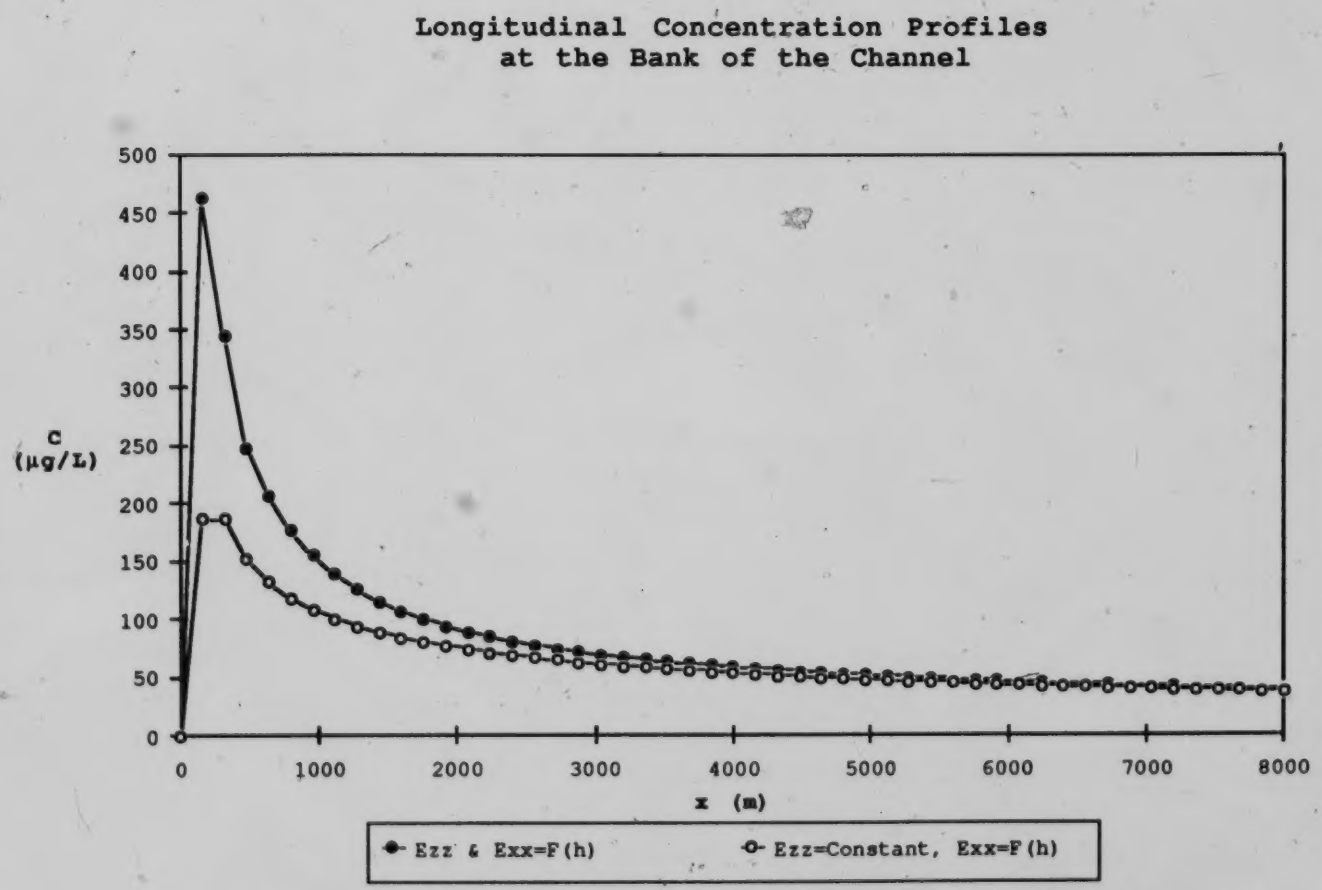


Figure 5.9 Concentration Profile along the Channel

boundary. However when E_{zz} was allowed to vary with the depth, which implies that E_{zz} is zero on the bank, the resulting contours are no longer orthogonal to the bank.

It is evident from the longitudinal profile and from the spreading rate plot as well as the close-up, that the use of $\overline{E_{zz}}$ results in the diffusion of the concentration in the bank. This implies that when a constant E_{zz} is used the maximum concentrations predicted are generally on the non-conservative side.

The specification of the E_{xx} is generally believed to be unimportant in the solution. This was reinforced by the analyses performed here. It is clearly apparent from the contour plots that different models of E_{xx} results in very little difference in the solution. This is also shown in the plots of the concentration on the bank, in Figure 5.10.

5.2 Application to River Plume Discharges

A plume study was done to model a river reach surveyed by Krishnappan and Lau (1982) on the Grand River near Kitchener, Ontario, Canada, to judge the performance of the numerical model in real river situation . A sketch of the river reach plan view, as well as a rough sketch of the cross-sections where data was measured is show in Figure 5.11.

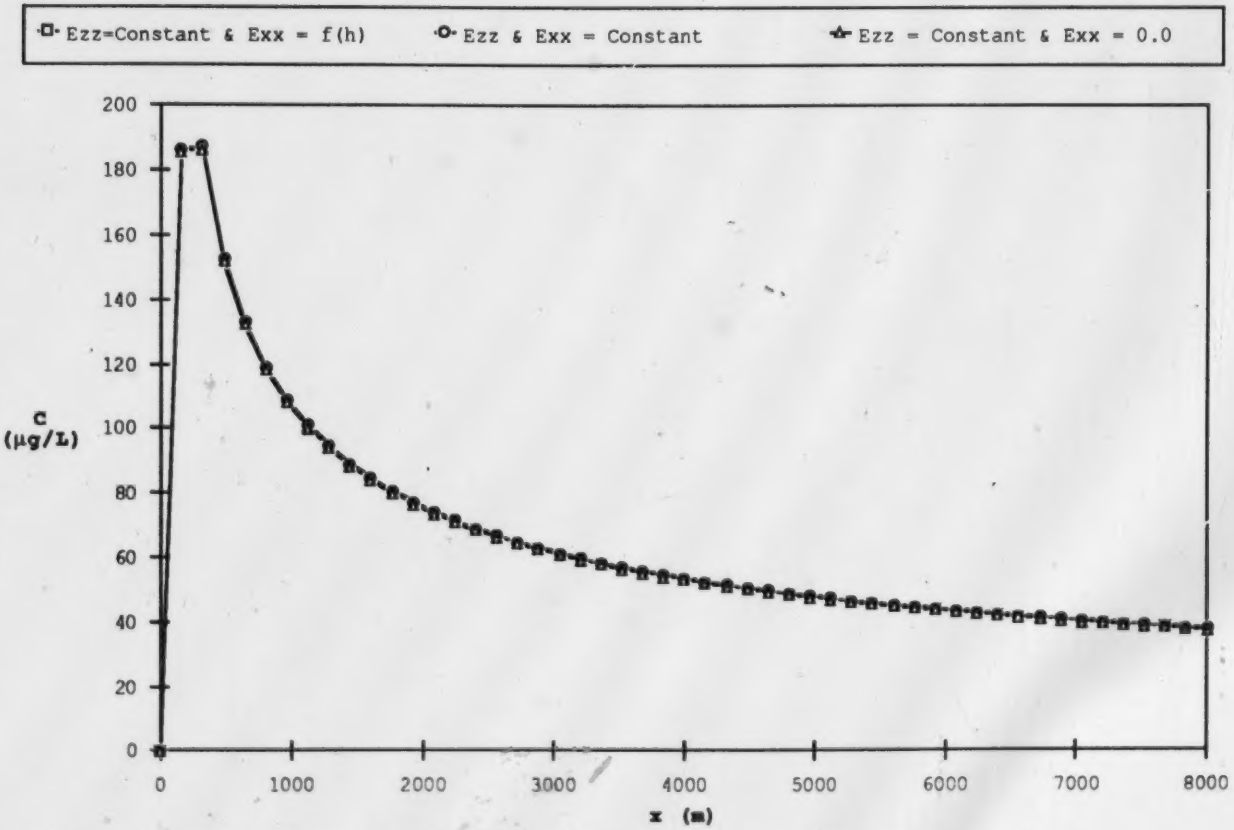


Figure 5.10 Longitudinal Concentration Profiles at the Bank of the Channel for cases b,c and d

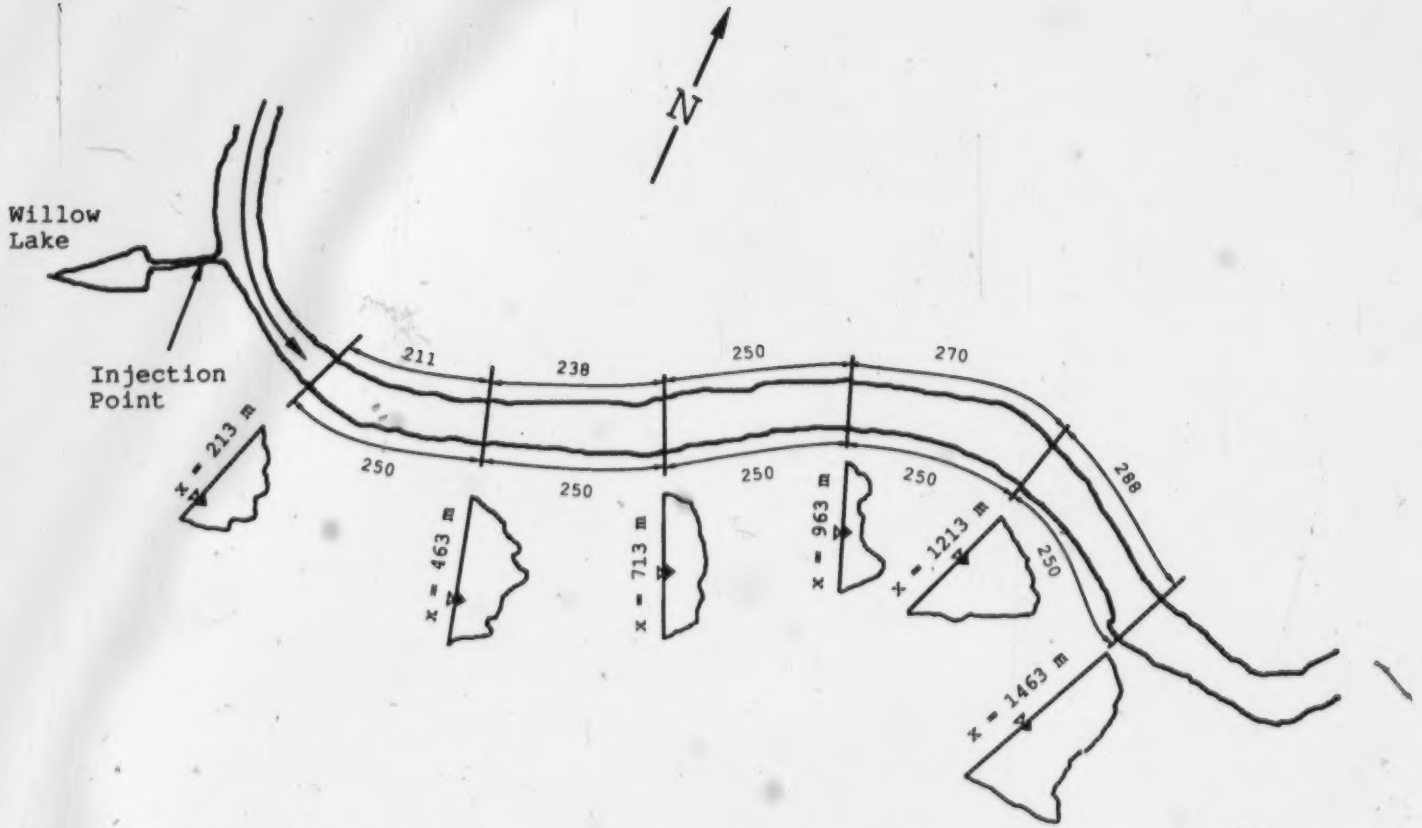


Figure 5.11 A Sketch of Grand River near Kitchener, Ontario, Canada (after Krishnappan and Lau 1982)

Krishnappan and Lau performed the plume test, where by a constant discharge of Rhodamine B dye was introduced into a tributary just downstream of Willow Lake. They subsequently measured the concentrations in the river at the cross-sections shown in Figure 5.11. They also measured the depth and a depth averaged velocity at each cross-section, at about 3.0 m intervals across the channel. The general river characteristics and pertinent data are reproduced in Table 5.1. The measured transverse diffusivity fitted the expression,

$$E_{zz} = 0.26\ h\ U_* \qquad (5.4)$$

according to Krishnappan and Lau.

The above river reach was discretized into a 40 long by 20 wide elements mesh, using quadratic mapping it interpolate the intermediate elements. The resulting mesh and velocity vectors are plotted in Figure 5.12. The eddy diffusivities were experimented with until the best solution was obtained. The values settled upon were given by the expressions,

$$E_{zz} = 0.25\ h\ U_* \text{ and} \qquad (5.5)$$

$$E_{xx} = 0.375\ h\ U_* . \qquad (5.6)$$

The solution for the plume was undertaken by imposing the measured concentration distribution at x = 213 m as a given boundary condition on the left side of the mesh. The numerical model was used to solve for the steady state solution which resulted in the concentration contours shown

TRANSECT NUMBER	DISTANCE FROM SOURCE (m)	CHANNEL WIDTH W (m)	AVERAGE DEPTH H (m)	MEAN VELOCITY U (m/s)	MEAN SHEAR VELOCITY U* (m/s)	FLOW RATE Q (m /s)	NON-DIMENSIONAL CHEZY COEFFICENT C*
1	213	49.3	,47	0.30	0.067	10.84	4.51
2	463	60.0	0.57	0.29	0.074	10.03	3.92
3	713	60.0	0.42	0.42	0.063	10.60	6.67
4	963	55.7	0.28	0.70	0.052	10.08	13.46
5	1213	57.0	0.65	0.29	0.079	10.70	3.67
6	1463	79.5	0.55	0.23	0.072	10.29	3.19

FOR EACH CROSS-SECTION A REPRESENTATIVE U* WAS USED
DEPTH WAS ALLOWED TO VARY ACROSS THE CHANNEL AND THEREFORE EVERY WHERE

$$E_{zz} = 0.25\ U^* h$$
$$E_{xx} = 0.38\ U^* h$$

Table 5.1 Hydraulic Data for the Grand River

in Figure 5.12. The concentration distribution across the channel at each cross-section is shown in Figure 5.13. As a comparison tool the concentration profiles were also plotted against the measured results in Figures 5.14 to 5.19.

The results from the numerical solution seem to be in agreement with the measured values. The discrepancies seen at x = 963 and x = 1213 are largely due to the fact that the channel reach is not straight. With a curved channel it is almost impossible to describe the behavior of the velocity field by any known method. The magnitude of the velocity vectors derived here are only representative at the given cross-sections. Due to the fact that the direction is not known, the velocity field used here is only a best estimate of the actual field. The model used here to interpolate the direction is only applicable to well behaved river reaches. In reaches such as this one where there is a substantial curvature the model has difficulties. The ability to accurately predict the measured concentration field is hampered by the lack of velocity direction information.

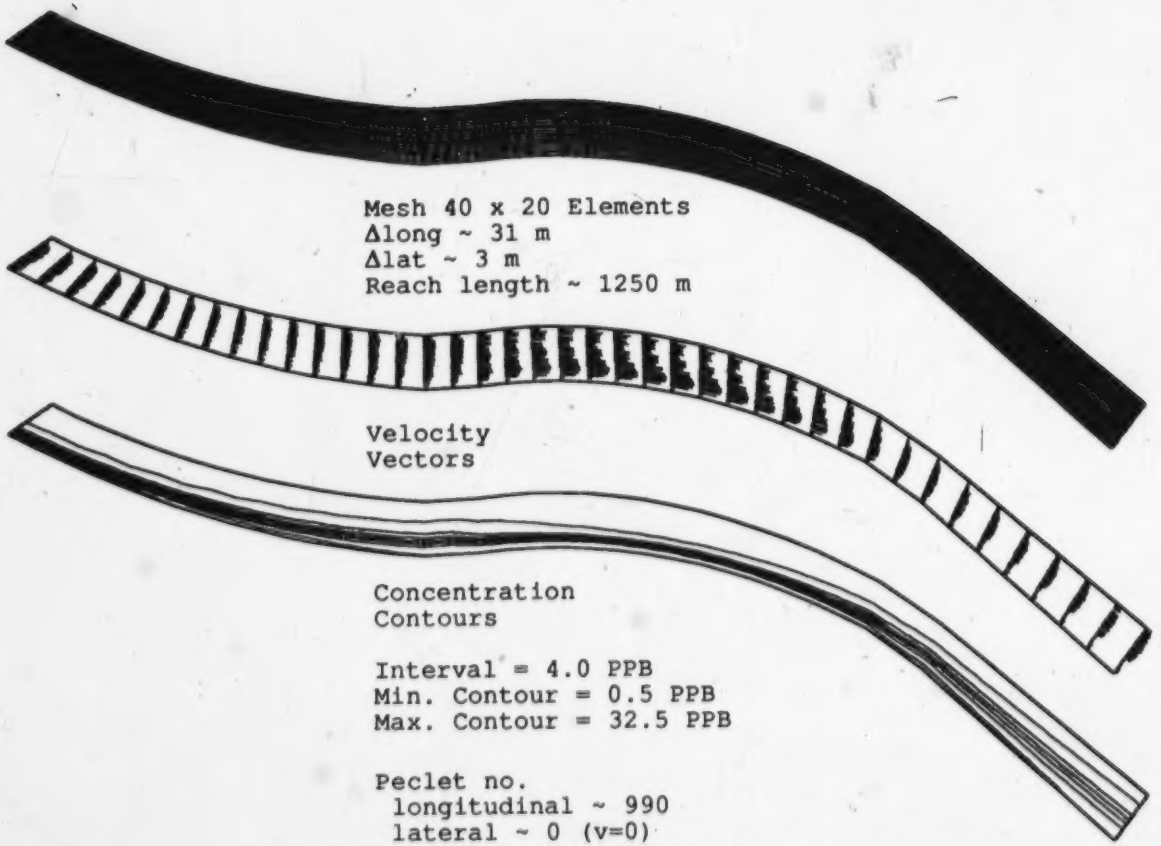


Figure 5.12 Mesh, Velocity, and Contours for Grand River

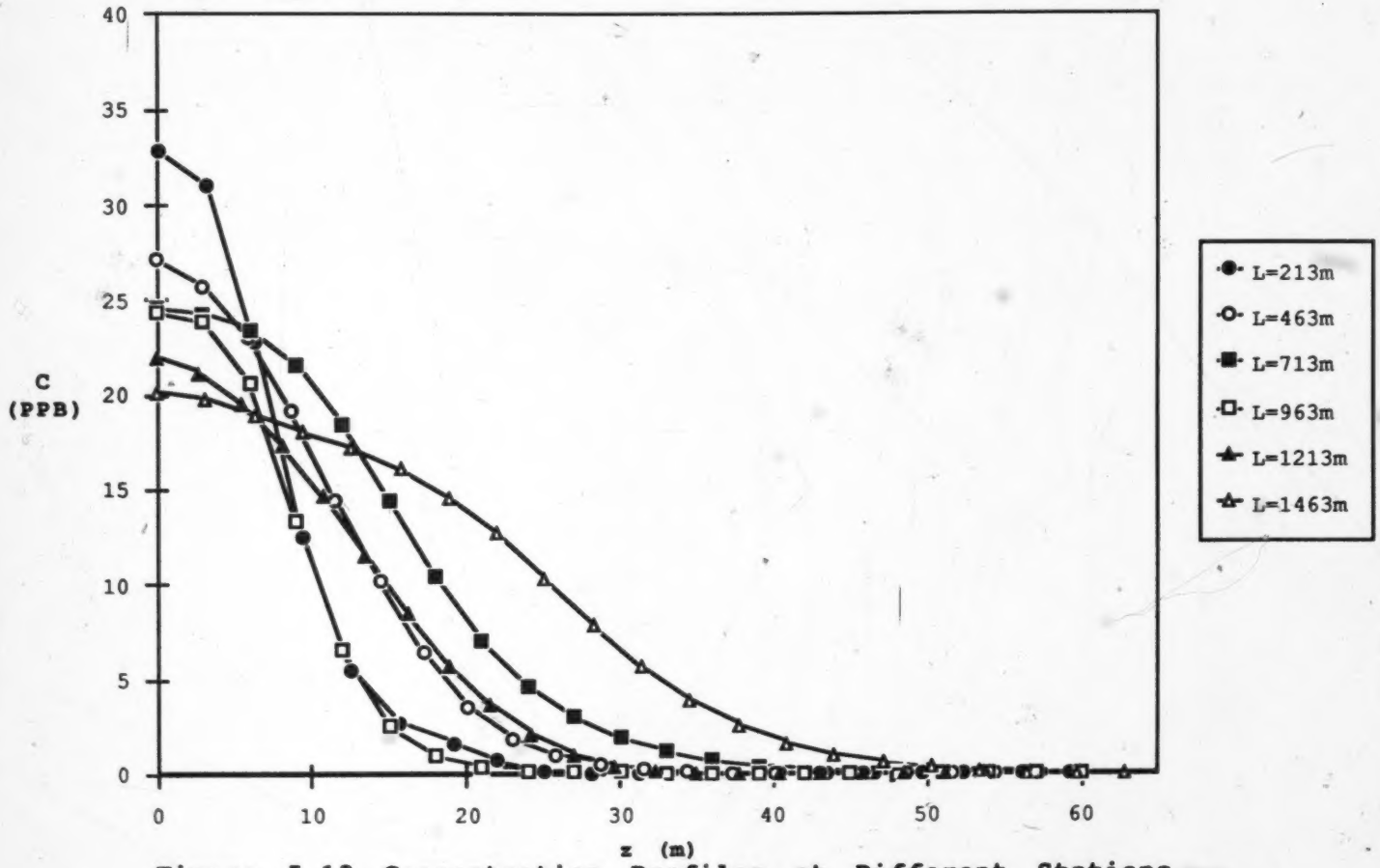


Figure 5.13 Concentration Profiles at Different Stations Grande River near Kitchener Ontario Canada

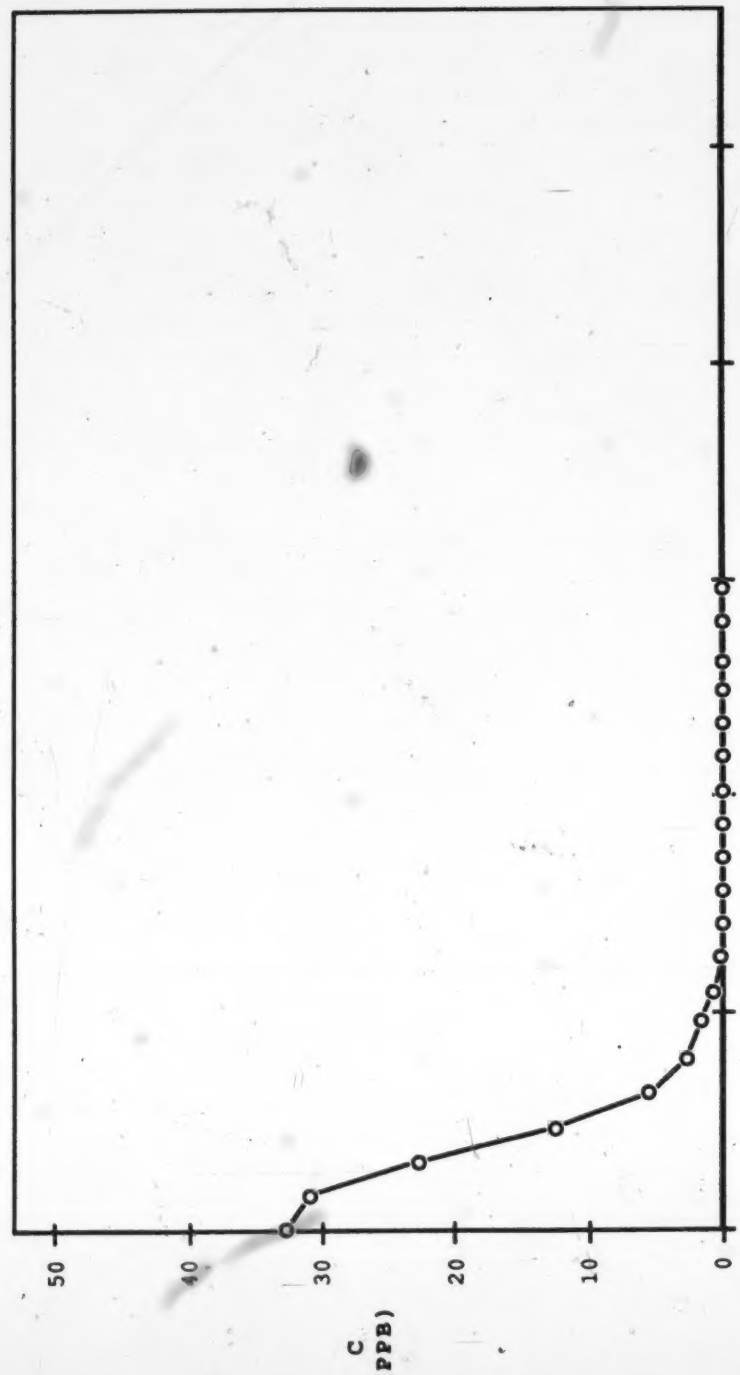


Figure 5.14 Concentration Profile at L = 213 m, Grand River

L = 463 m

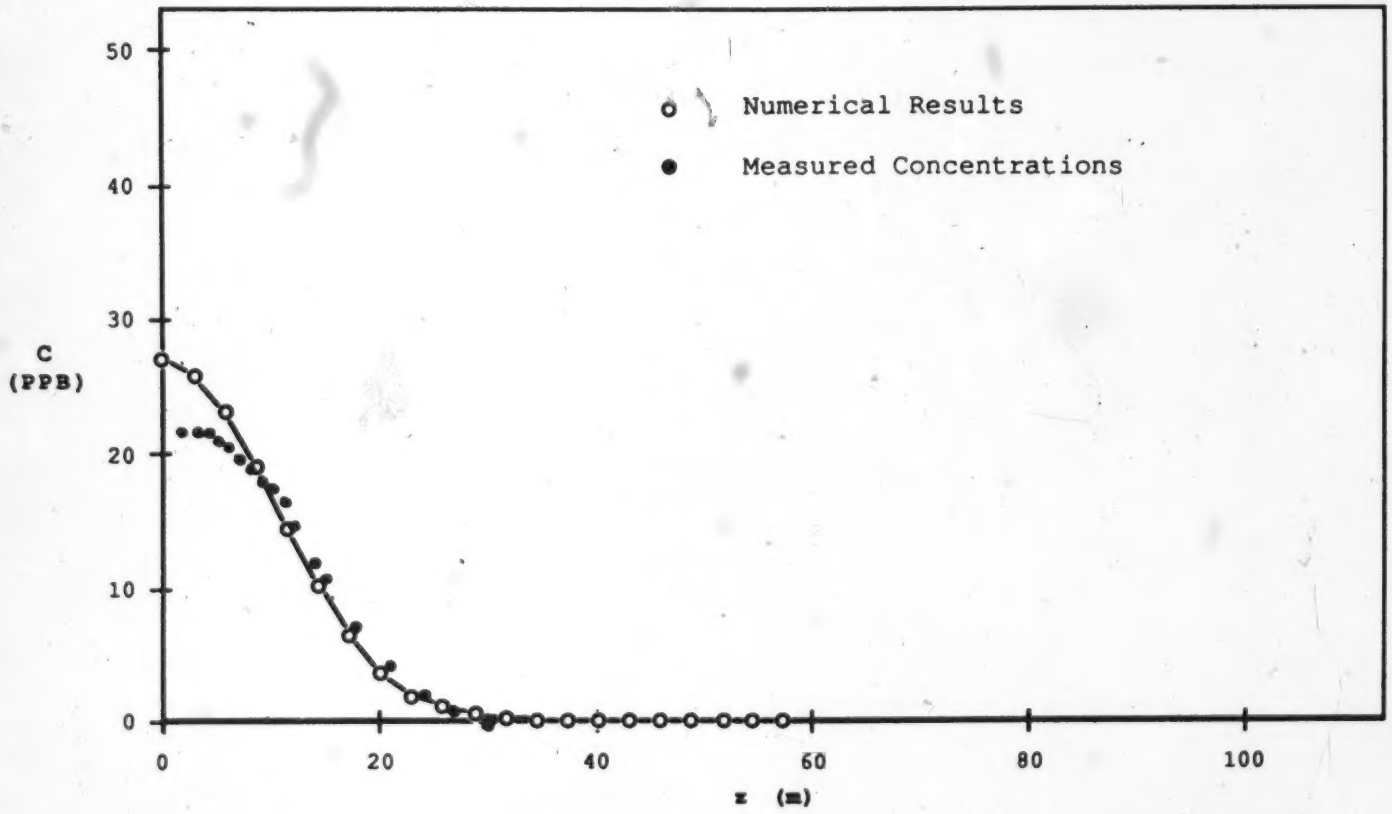


Figure 5.15 Concentration Profile at L = 463 m, Grand River

L = 713 m

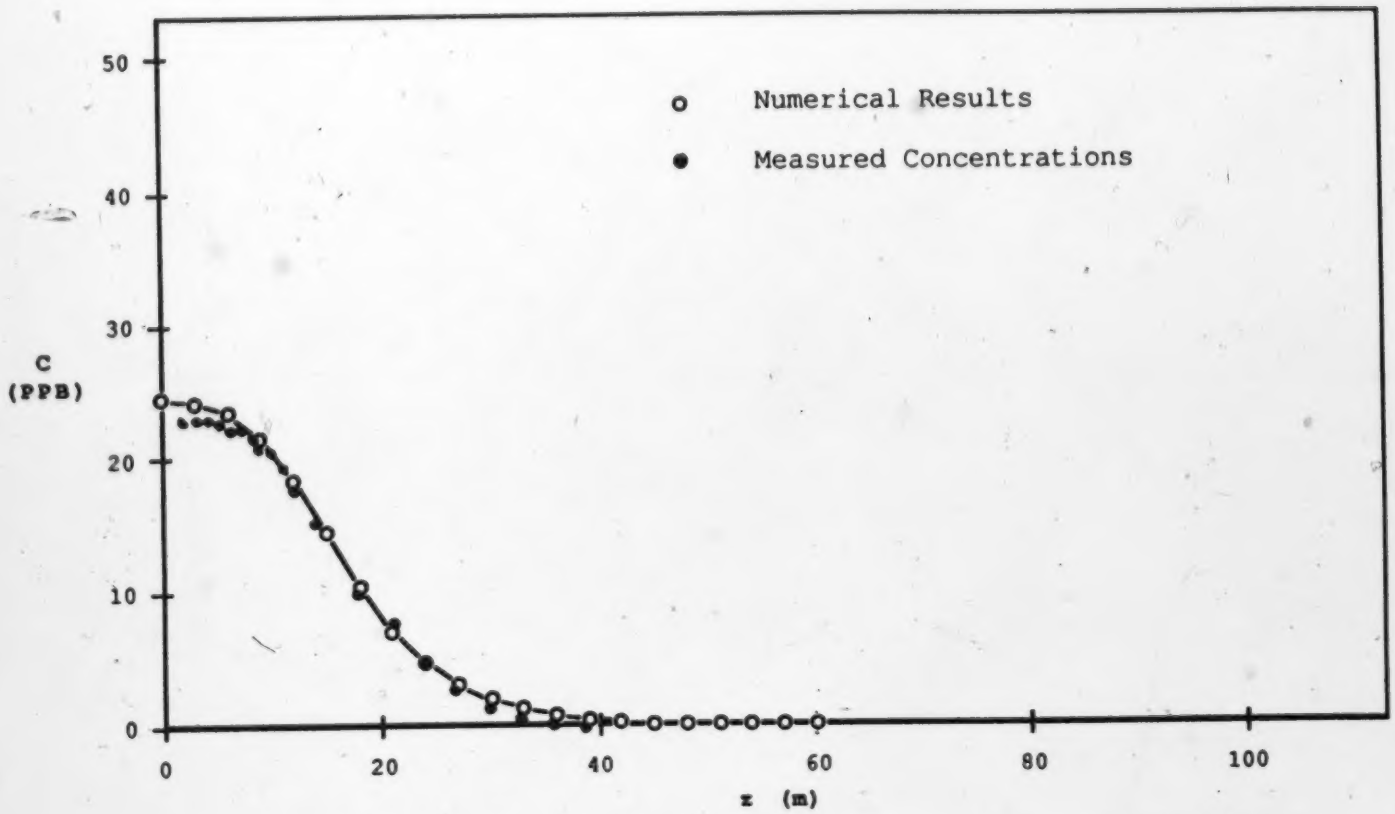


Figure 5.16 Concentration Profile at L = 713 m, Grand River

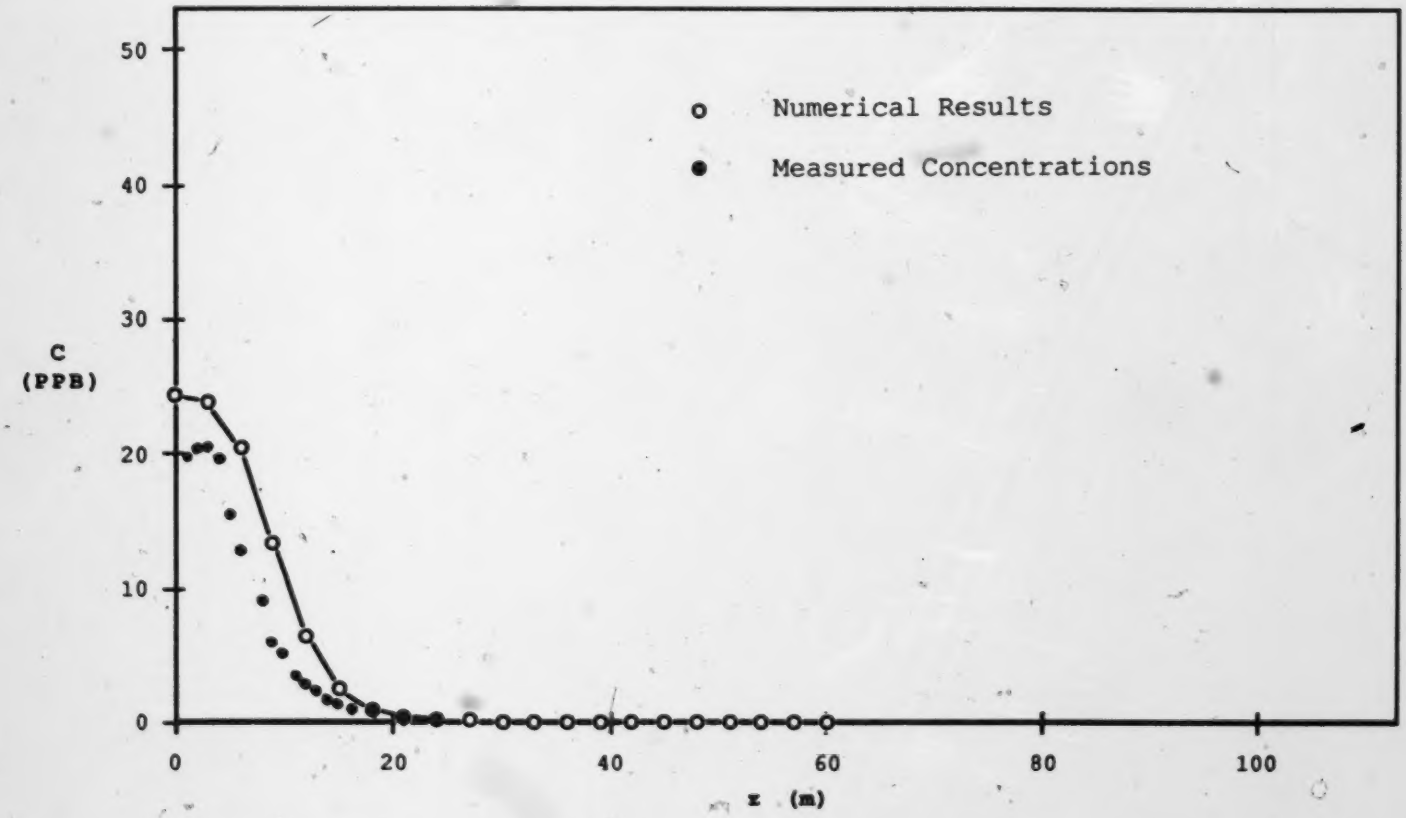


Figure 5.17 Concentration Profile at L = 963 m, Grand River

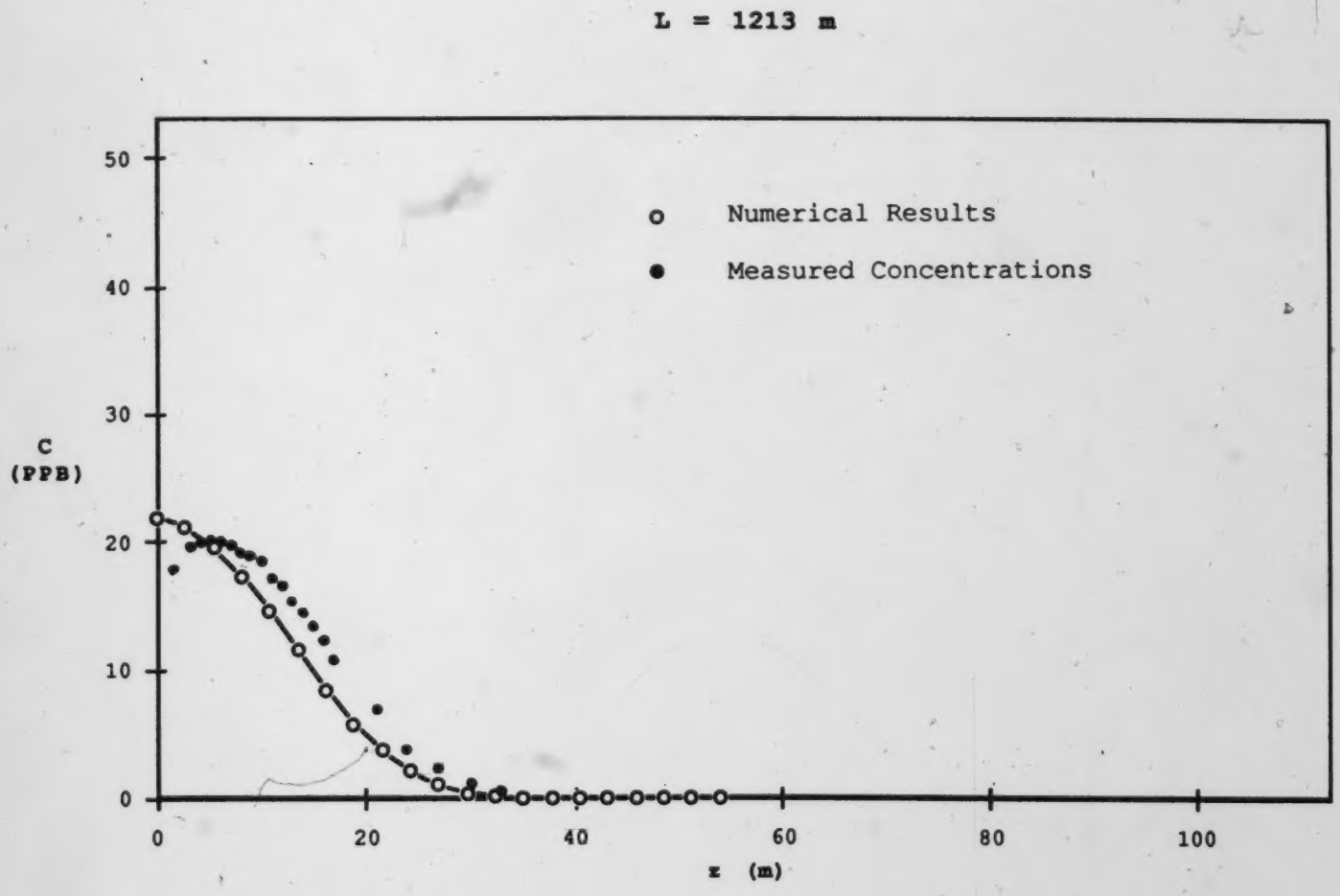


Figure 5.18 Concentration Profile at L = 1213 m, Grand River

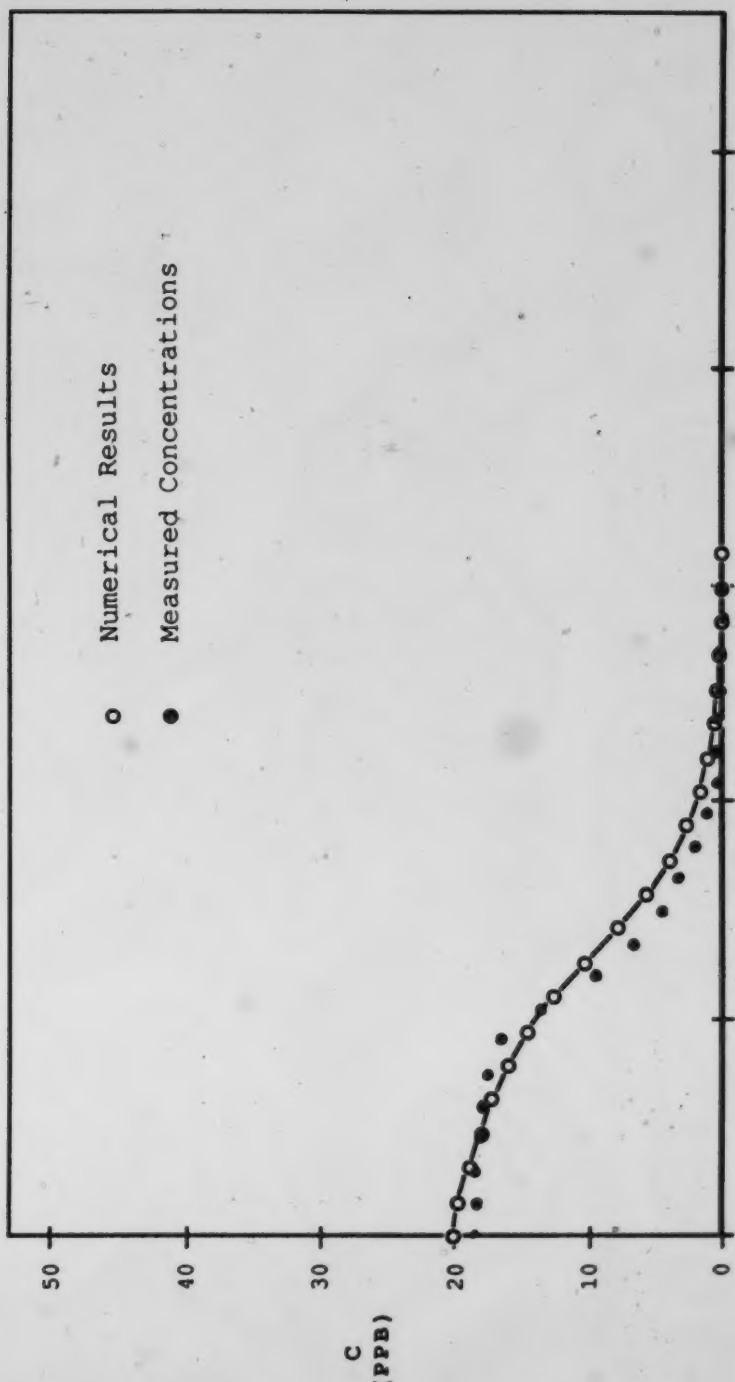


Figure 5.19 Concentration Profile at L = 1463 m, Grand River

5.2 Application to River Slug Tests

One of the most important concerns that the engineer is faced with is the accidental release of a contaminant into a river. The result of such a release is the possibility of large concentrations of pollutant entering the drinking water inlets situated on the river. To understand and predict the behavior, a 'slug test' is performed on most commercially used rivers. The ability to predict the behavior of such releases into the river by a numerical model is very desirable. However the physics of the processes involved in the predictive models is not yet fully understood.

An important component of the physics that is yet to be understood is the process of dispersion. Although there are models available such as the Elder model for logarithmic velocity profiles discussed in Section 2.1.4.1 (1959), they are limited to the final region. S. Beltaos(1980) also proposed a model to predict the dispersive behavior of a slug in the initial region which could reconstruct the skewed distribution found in reality. However his model basically amounted to an empirical fit to the processes observed, unrelated to any physical arguments.

This lack of physical understanding of the dispersion processes has made it impossible to reproduce the measured skewed distributions. However for the sake of application, the current finite element model was applied to such a case.

S. Beltaos at Alberta Research Council undertook a series of tests on the Athabasca River below Fort McMuarry. Among these was a test performed under an ice covered river reach. The detailed information on this particular test was reported in a Alberta Research Council Report (1978).

The reach considered in this report was discretized into a mesh 9 elements wide by 100 elements long. The discretized reach is shown in a plot in Figure 5.20. Included in Figure 5.20 is a plot of the depth averaged velocities derived from measurements and interpolation. A plot of the initial concentration slug is also shown in this figure.

A summary of the hydraulic data and the model used for the eddy diffusivity is shown in Table 5.2. Using the reach averaged velocity of 0.49 m/s, an average Δx = 118 m and a Courant number of 0.125, a time interval of 30 seconds was realized. An increased level of accuracy was needed due to the unsteady nature of the problem. Earlier investigations (Section 4.1) had shown that QUPG gave an adequately accurate solution to the unsteady advection of a pulse, thus QUPG was used to solve this problem. CUPG was not used due to a limitation on the memory available to solve the problem, .

The contour plots for the results at times 100 min., 200 min., and 300 min., are shown in Figure 5.21. A plot of the history of the concentration across the channel at x = 6330 m is shown in Figure 5.22.

The results show the potential of numerical solution to the unsteady problem in its ability to provide the engineer a wealth of data. The model allows the investigator to observe anomalies such as of pockets of relatively large concentrations attach to some banks as the one shown in Figure 5.21 on the left bank.

There exists a need to develop a physically sound analysis for the modelling of dispersion in rivers. Without this base, the advancement in the use of numerical solutions in aiding the engineer in assessing behavior of transient slugs in rivers is limited.

TRANSECT NUMBER	DISTANCE FROM SOURCE (m)	CHANNEL WIDTH W (m)	AVERAGE DEPTH H (m)	MEAN VELOCITY U (m/s)	E_long. Calcu. (m /s)	E_lat. Calcu. (m /s)
1	0	225.0	1.342	0.30	0.069	0.046
2	1900	166.5	2.327	0.35	0.135	0.090
3	3100	427.0	1.252	0.26	0.062	0.041
4	4300	288.0	1.555	0.33	0.073	0.049
5	6300	298.5	1.410	0.35	0.063	0.042
6	7800	130.0	2.595	0.41	0.168	0.112
7	9700	162.0	2.727	0.40	0.170	0.113
8	11800	288.0	1.836	0.28	0.091	0.061

AN AVERAGE U* WAS USED
DEPTH WAS ALLOWED TO VARY ACROSS THE CHANNEL AND THEREFORE EVERY WHERE

Slope = 0.0001444
Average U* = 0.037 m/s
Ezz = 0.58 U* h
Exx = 0.87 U* h

Table 5.2 Hydraulic Data for the Athabasca River below Fort McMurray (according to S. Beltaos,1978)

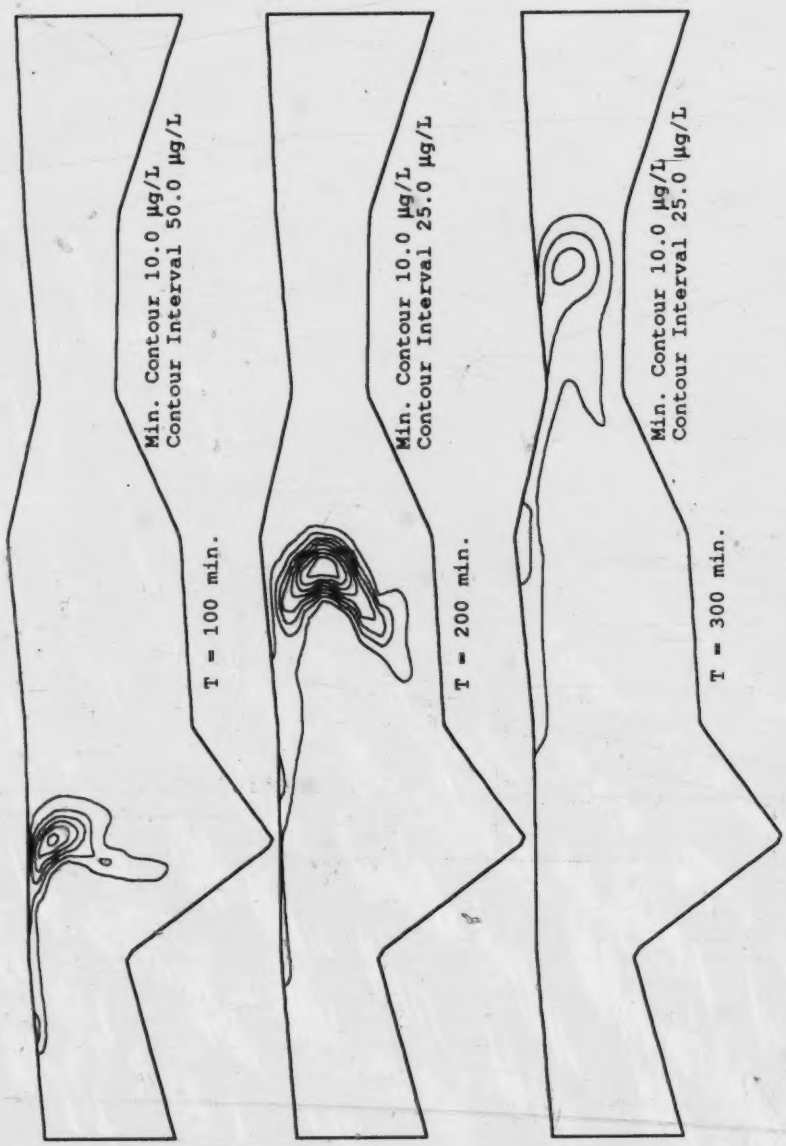

Min. Contour 10.0 µg/L
Contour Interval 50.0 µg/L
T = 100 min.
Min. Contour 10.0 µg/L
Contour Interval 25.0 µg/L
T = 200 min.
Min. Contour 10.0 µg/L
Contour Interval 25.0 µg/L
T = 300 min.

Figure 5.21 Concentration Contours for a Slug test

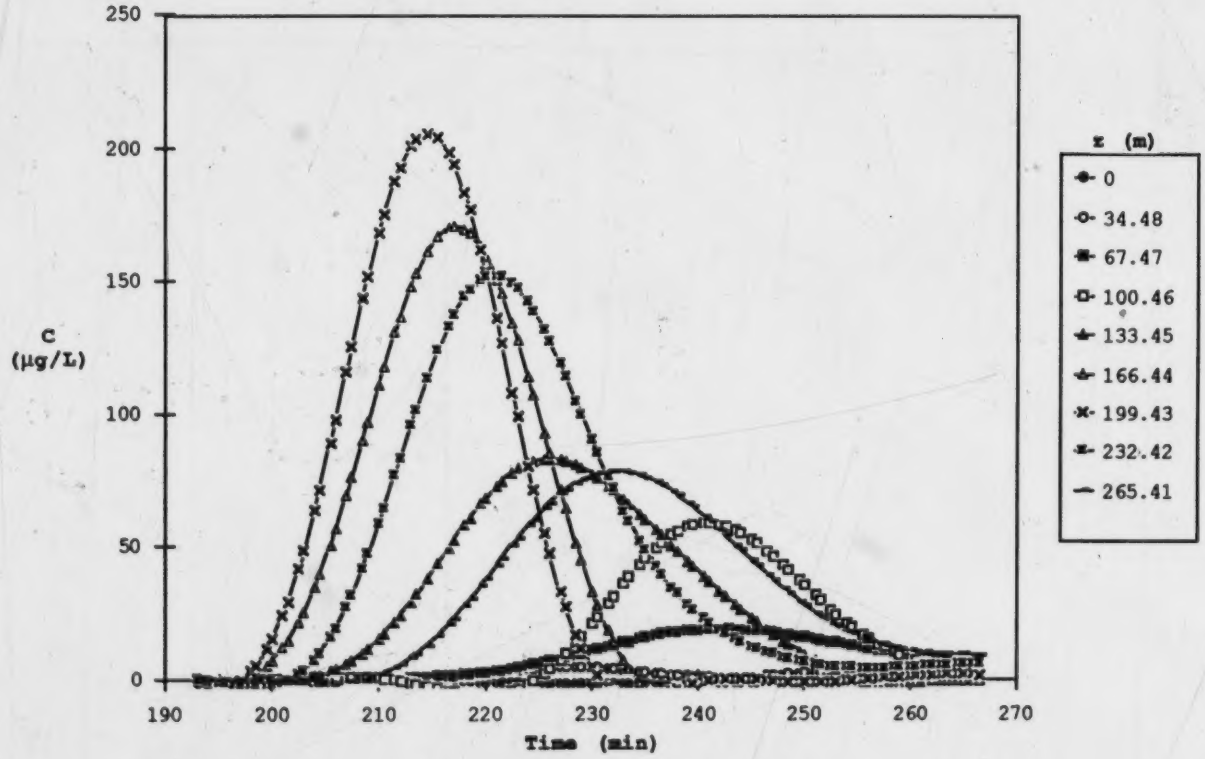


Figure 5.22 Computed Concentrations at x = 6360 m, Athabasca River

6. CONCLUSIONS

The use of the finite element method to solve the depth averaged pollutant equation gives the engineer an added degree of freedom in his ability to accurately present the picture of the behavior of pollutants in rivers.

The new upwinding elements, developed by Dr. P. M. Steffler, were found to be far superior than most of the standard finite difference schemes used in the modelling of the pollutant conservation equation. They were also found to be better than the standard linear finite elements. However the use of these elements has to be justified by a need to model flows with high Peclet numbers, as they require extra computational effort to solve.

The solutions of the two dimensional advection equation for a circular flow field, suggests that the elements are best suited for conditions where the direction of the flow is not known at the start. The ability of these elements to upwind in any desired direction is best realized by circulation problems. A situation like this exist in the modelling of effluent discharges in coastal waters, where the majority of the flow field is in circular motion and in an alternating direction. It is recommended that these new elements be applied to problems of outfall discharges in coastal waters.

The use of a variable eddy diffusivity in an idealized channel was found to produce a more realistic solutions than an average eddy diffusivity. The finite element model with the freedom to specify a variable eddy diffusivity, proved to be adequate in reproducing the measured concentrations for the Grand River reach.

The need to develop a firm physical understanding of the dispersion processes in a river slug test, was pointed out by the results of the numerical model applied to a 'slug' test. The solution presented by the model showed the large potential it has in providing the engineer with a wealth of information.

REFERENCES

Baker A. J., 1983, Finite Element Computational Fluid Mechanics. Hemisphere Publishing Corporation, New York, N.Y.

Beltaos S., 1978, Transverse Mixing in Natural Streams, Report No. SWE78-1. Alberta Research Council, Edmonton, Alberta.

Beltaos S., 1980, "Longitudinal Dispersion in Rivers". ASCE Journal of the Hydraulics Division. Vol. 106(HY10), pp. 1607-1625.

Elder J. W., 1959, "The dispersion of marked fluid in turbulent shear flow", Journal of Fluid Mechanics, Vol. 5, pp. 544-560.

Elhdi N., Harrington A., Hill I., Lau Y. L., Krishnappan B. G., 1984, "River mixing-A state-of-art report", Canadian Journal for Civil Engineer, Vol. 11, pp. 585-609.

Fisher H. B., List E. J., Koh R. C., Berger G. N., and Brooks N. H., 1979, Mixing in Inland and Coastal Waters. Academic Press, New York, N. Y.

Harden T. O. and Shen H. T., 1979, "Numerical simulation of mixing in natural rivers", ASCE Journal of the Hydraulic Division, Vol. 105(HY4), pp. 393-408.

Krishnappan B. G. and Lau Y. L., 1977, "Transverse mixing in meandering channels with varying bottom topography", Journal of Hydraulic Research, Vol 15(4), pp. 351-371.

Krishnappan B. G. and Lau Y. L., 1981, "Modelling transverse mixing in natural streams". ASCE Journal of the Hydraulic Division, Vol. 107(HY2), pp. 209-226.

Krishnappan B. G. and Lau Y. L., 1983, User's manual for model RIVMIX, National Water Research Institute, Environment Canada, Burlington, Ontario.

Lapidus L. and Pinder G.F., 1982, Numerical Solution of Partial Differential Equations in Science and Engineering, John Wiley and Sons, Toronto, Canada, 677 pages.

Leimkuhler W., Connor J., Wang J., Christodouluou G. and Sundgren S., 1975, "Two Dimensional Finite Element Dispersion Model". ASCE Symposium on Modeling Techniques, San Fransico, California, pp. 1467-1486.

Leonard B. P., 1979, "A Stable and Accurate convective modelling procedure based on quadratic Upstream Interpolation", J. Comp. Mtd. Appl. Mech. and Eng., Vol 19, pp. 59-98.

Putz G., 1984, TRSMIX Users Manual, Environmental Engineering Technical Report 84-2, University of Alberta, Edmonton, Alberta.

Rajaratnam N., 1976, Turbulent Jets, Developements in Water Science, Elsevier Scientific Publishing Company, New York, N. Y., 304 pages.

Sauvaget P., 1985, "Dispersion in Rivers and Coastal Waters - 2. Numerical Computation of Dispersion", Developments in Hydraulic Engineering, Vol 3, Elsevier Applied Science Publishers, New York, N.Y., pp. 39-78.

Sayre W. W. and Chang F. M., 1968, A Laboratory Investigation of the Open Channel Dispersion Process for Dissolved, Suspended and Floating Dispersants, U. S. Geological Survey Professional Paper 433-E.

Sayre W. W., 1973, "Natural mixing processes in Rivers", Environmental impact on Rivers, Edited by H. W. Shen, H. W. Shen, P. O. Box 606, Fort Collins, CO, U. S. A., Chapter 6.

Steffler P. M., 1988, "Upwind Basis Finite Elements for Convective Dominated Problems", To be Published in the International Journal for Numerical Methods in Fluids.

Stone H. L. and Brian P. L. T., 1963, "Numerical Solution of Convective Transport Problems", AIChE Journal, Vol. 9, pp. 681-688.

Taylor G. I., 1921, "Diffusion by Continuous Movements", the Proceedings of the London Mathematical Society, Vol. 20, pp. 196-212.

Yotsukura N. and Sayre W. W., 1976, "Transverse mixing in Natural Channels", Water Resources Research, Vol. 12(4), pp. 695-704.

APPENDIX 1

Program CDFEM Listing

Files Included Are

	Page
fe.h	134
femain.c	137
feio.c	140
fecore.c	149
feinitial.c	165
fecom.c	167
fetest.c	180
fe_nupf.c	194
fegauss.c	209
festuff.c	221
festuff1.c	227
festuff2.c	241
festuff3.c	249
festuff4.c	254
festuff5.c	258
Example Input file for Generation of a River Mesh	260
Example Output of a Mesh	263

```
1   #include "math.h"
2   #include "stdio.h"
3
4   long FreeMem() ;      /*     NON STANDARD FUNCTION       */
5
6   char   *calloc(), *lmalloc() ; /*  NON STANDARD FUNCTION       */
7
8   #define  MNSF         36
9   #define  MRNSF        4
10  #define  MDOF         144
11  #define  MGPTS        16
12  #define  NDIMS        2
13  #define  NPARAMS      8
14  #define  NBPARAMS     2
15  #define  NVARS        1
16  #define  NKEQNS       20
17  #define  MNXS         40
18
19  struct shapefuncs {
20         int    dof ;
21         double f[MNSF] ;
22         double        dfdr[MNSF] ;
23         double dfds[MNSF] ;
24         double dfdt[MNSF] ;
25         } ;
26
27  struct govfuncs {
28         int    dof ;
29         double detJ ;
30         double w ;
31         double p[NPARAMS] ;
32         double f[MNSF] ;
33         double dfdx[MNSF] ;
34         double dfdy[MNSF] ;
35         double dfdz[MNSF] ;
36         } ;
37
38  struct gausspts {
39         double x ;
40         double y ;
41         double z ;
42         double w ;
43         } ;
44
45  struct node  {
46         int           n ;
47         int           i ;
48         int           ui ;
49         int           fxc    ;
50         double        x[NDIMS]      ;
51         double        u[NVARS]      ;
52         float         p[NPARAMS]    ;
53         float         f[NDIMS]      ;
54         struct node   *nextnp       ;
55         } ;
56
57  struct element  {
58         int           n ;
59         int           vtype ;
60         int           gtype ;
61         int           nnds ;
62         struct node   *nps[MNSF];
63         float         p[NPARAMS] ;
```

```
64          double          *matrices ;
65          float           elinfo ;
66          struct element          *nextelp ;
67          } ;
68
69    struct belement  {
70          int             n ;
71          int             vtype ;
72          int             gtype ;
73          int             nnds ;
74          struct node     *nps[MRNSF] ;
75          double          p[NBPARAMS] ;
76          int             bcs[NVARS] ;
77          struct belement         *nextbelp ;
78          } ;
79
80    struct control {
81          int     trans ;
82          int     dims ;
83          int     vars ;
84          int     params ;
85          int     bparams ;
86          int     Keqns[NKEQNS] ;
87          int     nodes ;
88          int     elms ;
89          int     belms ;
90          int     frnds ;
91          int     ukns ;
92          int     sym ;
93          } ;
94
95    struct eqnset {
96          double *Kp ;
97          double *Lp ;
98          double *Fp ;
99          long   *diagp ;
100         int    neqns ;
101         } ;
102
103   struct pointers {
104         struct node             *N ;
105         struct element          *El ;
106         struct belement *B ;
107         struct node     **iptrs ;
108         double          *S ;
109         double          *M ;
110         }   ;
111
112   struct elmpointers {
113         double *K ;
114         double *F ;
115         double *S ;
116         double *M ;
117         int    n  ;
118         }  ;
119
120   struct fxnodes {
121         int    i ;
122         int    j[MDOF] ;
123         double k[MDOF] ;
124         double f ;
125         } ;
126
```

```
127  struct transient {
128         int    nsteps ;
129         double dtfac ;
130         double t ;
131         double dt ;
132         double theta ;
133         } ;
134
135  struct RegMesh {
136         int            nx ;
137         int            ny ;
138         int            nbx ;
139         int            nby ;
140         int            elsinbx[MNSF] ;
141         int            elsinby[MNSF] ;
142         int            eltype ;
143         int            maptype ;
144         struct control       con ;
145         struct blockmap      *firstmap ;
146         } ;
147
148  struct blockmap {
149         float  xnodes[MNSF] ;
150         float  ynodes[MNSF] ;
151         float  h[MNSF] ;
152         float  u[MNSF] ;
153         float  v[MNSF] ;
154         float  E_x[MNSF] ;
155         float  E_y[MNSF] ;
156         float  E_xy[MNSF] ;
157         float  E_yx[MNSF] ;
158         } ;
159
160  struct cross_section {
161         float                x[MNXS] ;
162         float                z[MNXS] ;
163         float                h[MNXS] ;
164         float                u[MNXS] ;
165         float                v[MNXS] ;
166         float                E_long[MNXS] ;
167         float                E_lat[MNXS] ;
168         struct cross_section        *next_x ;
169         } ;
```

femain.c

```
#include      "fe.h"
#include      "time.h"


char  *strcat(), *strncat();

main()
{
        char          ain ;
        long          t_bytes ;
        int           nin, errcode, i ;
        long int      t1, t2 ;
        void          MaxApplZone() ;/*NON STANDARD FUNCTION */

        MaxApplZone() ;

        init_gp() ;

        printf(" Input command (s,t,i,n,m,l,b,o,p,t,c,r,y) and number\n") ;
        ain = 'b' ;

        while(ain != 'q'){
                time(&t1) ;
                t_bytes = FreeMem() ;/*     NON STANDARD FUNCTION */
                printf(" Total Memory Available =%ld\n",t_bytes) ;
                printf("> ");
                scanf(" %c %d",&ain,&nin) ;
                switch (ain) {

                        case 's' :
                                errcode = steady_state(nin,1) ;
                                break ;
                        case 'i' :
                                errcode = input(nin) ;
                                break ;
                        case 'n' :
                                errcode = test_mesh(nin) ;
                                break ;
                        case 'm' :
                                errcode = test_map(nin) ;
                                break ;
                        case 'l' :
                                errcode = list_vars(nin) ;
                                break ;
                        case 'b' :
                                errcode = set_bc(nin) ;
                                break ;
                        case 'o' :
                                errcode = output(nin) ;
                                break ;

                        case 'p' :
                                errcode = special_out(nin) ;
                                break ;
                        case 't' :
                                errcode = test_trans(nin) ;
                                break ;
                        case 'c' :
                                errcode = test_contour(nin) ;
                                break ;
                        case 'r' :
                                errcode = test_real_stream(nin) ;
```

```
64                              break ;
65                      case 'y' :
66                              errcode = test_analytical(nin) ;
67                              break ;
68                  }
69                  time(&t2) ;   /*     NON STANDARD FUNCTION */
70                  printf(" Time = %s",ctime(&t2)) ;
71                              /*     NON STANDARD FUNCTION */
72          }
73  }
74
75  FILE    *get_fptr(code)
76  int             code ;
77
78  {
79          FILE    *fp, *fopen() ;
80          char    s[63] ;
81
82          if(code == 1) {
83                  printf("\n Input Data File Name  ") ;
84                  scanf(" %s",s) ;
85                  fp = fopen(s,"r") ;
86          }
87          else
88                  fp = stdin ;
89          return( fp ) ;
90  }
91
92  FILE    *put_fptr(code)
93  int             code ;
94
95  {
96          FILE    *fp, *fopen() ;
97          char    s[63] ;
98
99
100         if(code == 1) {
101                 printf("\n Input Output File Name  ") ;
102                 scanf(" %s",s) ;
103                 fp = fopen(s,"w") ;
104         }
105         else if(code == 2)
106                 fp = fopen("CDFEM1.out","w") ;
107         else if(code == 3)
108                 fp = fopen("CDFEM2.out","w") ;
109         else if(code == 4)
110                 fp = fopen("CDFEM3.out","w") ;
111         else if(code == 5)
112                 fp = fopen("CDFEM4.out","w") ;
113         else if(code == 6)
114                 fp = fopen("CDFEM5.out","w") ;
115         else if(code == 7)
116                 fp = fopen("CDFEM6.out","w") ;
117         else
118                 fp = stdin ;
119
120         return( fp ) ;
121 }
122
123 int             StartMes(astr,bstr,cstr,dstr)
124 char            *astr, *bstr, *cstr, *dstr ;
125
126 {
```

```
127         printf("%s\n%s\n%s\n%s\n",astr,bstr,cstr,dstr) ;
128         return(0);
129     }
130
131     int             UpDateMes(astr,bstr, cstr, dstr)
132     char            *astr, *bstr, *cstr, *dstr ;
133
134     {
135         printf("%s\n", dstr) ;
136         return(0);
137     }
138
139     int             EndMes()
140
141     {
142         return(0);
143     }
144     int             InsertMes(astr)
145     char            *astr ;
146
147     {
148         printf("%s\n",astr) ;
149         return(0);
150     }
151
```

```
1   #include "fe.h"
2   #include "ctype.h"
3
4   extern struct control              N ;
5   extern struct pointers             gp;
6   extern struct transient     tvals ;
7   extern struct RegMesh              Mesh ;
8
9   int                        Nndso, Nelmso, Nbelmso, Mesh_code ;
10
11  int           input(nin)
12  int           nin ;
13
14  {
15        int     err,i, fer ;
16        FILE    *fptr, *get_fptr() ;
17
18        if((fptr = get_fptr(nin) ) != NULL) {
19              if( (err = get_control(fptr)) == 0) {
20                    put_control(stdout) ;
21                    for(i=0;i<N.nodes;i++)
22                          get_node(fptr,i) ;
23                    for(i=0;i<N.elms;i++)
24                          get_elm(fptr,i) ;
25                    for(i=0;i<N.belms;i++)
26                          get_belm(fptr,i) ;
27                    if(fptr != stdin) fclose(fptr) ;
28                          set_bvalues() ;
29              }
30              else
31                    printf(" Control Variable Error %d\n",err) ;
32
33              if((fer = ferror(fptr))!= NULL || (fer != 0) ){
34                    printf(" Read error in routine Input \n Error ");
35                    clearerr(fptr) ;
36                    if(fptr != stdin) fclose(fptr) ;
37                    return(fer) ;
38              }
39        }
40        else {
41              printf(" Unable to open that file\n") ;
42              err = -1 ;
43              return(err);
44        }
45  }
46
47  int     get_control(f)
48  FILE    *f ;
49
50  {
51        int     i, j, k, in, ii ;
52        long    bigint ;
53        char    *calloc(), *lmalloc() ; /*  NON STANDARD FUNCTION       */
54
55        j = 0 ;
56        for(i=0;i<1;i++) {
57              j++ ;
58              N.trans = get_int(f) ;
59              if(N.trans < 0 || N.trans > 1) break ;
60              j++ ;
61              if(N.trans == 1)
62                    get_trans(f) ;
63              N.dims = get_int(f) ;
```

```
64                  if(N.dims < 1 || N.dims > NDIMS) break ;
65                  j++ ;
66                  N.vars = get_int(f) ;
67                  if(N.vars < 1 || N.vars > NVARS) break ;
68                  j++ ;
69                  for(ii=0;ii<((N.vars+1)*N.vars);ii++)
70                        N.Keqns[ii] = get_int(f) ;
71                  j++ ;
72                  N.params = get_int(f) ;
73                  if(N.params < 0 || N.params > NPARAMS) break ;
74                  j++ ;
75                  N.bparams = get_int(f) ;
76                  if(N.bparams < 0){
77                        Mesh_code = -1 ;
78                        N.bparams *= -1 ;
79                  }
80                  else
81                        Mesh_code = 1 ;
82                  if(N.bparams < 0 || N.bparams > NBPARAMS) break ;
83                  j++ ;
84                  if(gp.N != NULL) {
85                        clear_data() ;
86                        free(gp.N) ;
87                        free(gp.iptrs) ;
88                  }
89                  N.nodes = get_int(f) ;
90                  Nndso = N.nodes ;
91                  bigint = (5 * N.nodes)/4 * sizeof(struct node *) ;
92                  if(N.nodes < 2 || (gp.iptrs = (struct node **) lmalloc(bigint))
93    == NULL) break ;
94                  bigint = (long int)N.nodes *(long int)sizeof(struct node) ;
95                  if(N.nodes < 2 || (gp.N = (struct node *) lmalloc(bigint)) ==
96    NULL) break ;/*      NON STANDARD FUNCTION       */
97                  j++ ;
98                  N.elms = get_int(f) ;
99                  Nelmso = N.elms ;
100                 bigint = (long int)N.elms * (long int)sizeof(struct element) ;
101                 if(gp.El != NULL)
102                       free(gp.El) ;
103                 if(N.elms < 1 || (gp.El = (struct element *) lmalloc(bigint)) ==
104   NULL) break ;/*      NON STANDARD FUNCTION       */
105                 j++ ;
106                 N.belms = get_int(f) ;
107                 Nbelmso = N.belms ;
108                 bigint =(long int) N.belms *(long int) sizeof(struct belement) ;
109                 if(gp.B != NULL)
110                       free(gp.B) ;
111                 if(N.belms < 1 || (gp.B = (struct belement *) lmalloc( bigint))
112   == NULL) break ;/*   NON STANDARD FUNCTION       */
113
114                 if(Mesh_code == -1 )
115                       get_mesh_info(f) ;
116
117                 printf(" Memory left = %ld\n",FreeMem()) ;/*     NON STANDARD
118   FUNCTION      */
119
120                 j = 0 ;
121           }
122           return(j) ;
123     }
124
125     int     get_mesh_info(f)
126     FILE    *f ;
```

```
127     {
128             Mesh.nx = get_int(f) ;
129             Mesh.nbx = get_int(f) ;
130             Mesh.ny = get_int(f) ;
131             Mesh.nby = get_int(f) ;
132             return(0) ;
133     }
134
135     int     get_trans(f)
136     FILE    *f ;
137     {
138             double get_dbl() ;
139
140             tvals.nsteps = get_int(f) ;
141             tvals.dtfac = get_dbl(f) ;
142             tvals.t = get_dbl(f) ;
143             tvals.dt = get_dbl(f) ;
144             tvals.theta = get_dbl(f) ;
145             return(0) ;
146     }
147
148
149     int     clear_data()
150
151     {
152             int             i ;
153             struct node     *np1, *np2 ;
154             struct element          *elp1, *elp2 ;
155             struct belement *belp1, *belp2 ;
156
157             np1 = gp.N ;
158             for(i=0;i<N.nodes;i++) {
159                     np2 = np1->nextnp ;
160                     if(i >= Nndso)
161                             free(np1) ;
162                     np1 = np2 ;
163             }
164             belp1 = gp.B ;
165             for(i=0;i<N.belms;i++) {
166                     belp2 = belp1->nextbelp ;
167                     if(i >= Nbelmso)
168                             free(belp1) ;
169                     belp1 = belp2 ;
170             }
171             elp1 = gp.El ;
172             for(i=0;i<N.elms;i++) {
173                     elp2 = elp1->nextelp ;
174                     if(elp1->matrices != NULL)
175                             free(elp1->matrices) ;
176                     if(i >= Nelmso)
177                             free(elp1) ;
178                     elp1 = elp2 ;
179             }
180             return(0) ;
181     }
182
183     int     get_node(f,i)
184     FILE    *f ;
185     int     i ;
186
187     {
188             int             j  ;
189             double          get_dbl() ;
```

```
190         struct node   *nodep ;
191
192         nodep = gp.N + i ;
193         nodep->n = get_int(f)  ;
194         nodep->i = i ;
195         nodep->fxc = 0 ;
196         gp.iptrs[i] = nodep ;
197         for(j=0;j<N.dims;j++)
198               nodep->x[j] = get_dbl(f) ;
199         for(j=0;j<N.params;j++)
200               nodep->p[j] = get_dbl(f) ;
201         for(j=0;j<N.vars;j++)
202               nodep->u[j] = get_dbl(f) ;
203         nodep->nextnp = nodep + 1 ;
204         return(0) ;
205   }
206
207   int     get_elm(f,i)
208   FILE    *f ;
209
210   {
211         int           j, jj, nnds, n, rc ;
212         struct element  *elmntp ;
213         struct node   *np ;
214
215         elmntp = gp.El + i ;
216         rc = 0 ;
217         elmntp->n = get_int(f) ;
218         elmntp->vtype = get_int(f);
219         elmntp->gtype = get_int(f);
220         elmntp->nnds =  nsf(elmntp->vtype) ;
221         nnds = nsf(elmntp->gtype);
222         if(nnds > elmntp->nnds) elmntp->nnds = nnds ;
223         if(elmntp->vtype == 228)
224               elmntp->nnds = 6 ;
225         if(elmntp->vtype == 229)
226               elmntp->nnds = 16 ;
227         if(elmntp->vtype == 233)
228               elmntp->nnds = 8 ;
229         if(elmntp->vtype == 230)
230               elmntp->nnds = 6 ;
231         if(elmntp->vtype == 239)
232               elmntp->nnds = 36 ;
233         if(elmntp->vtype == 129)
234               elmntp->nnds = 4 ;
235         if(elmntp->vtype == 139)
236               elmntp->nnds = 6 ;
237         for(j=0;j<elmntp->nnds;j++) {
238               if((n = get_int(f)) >= 0) {
239                     np = gp.N ;
240                     for(jj=0;jj<N.nodes;jj++) {
241                           if(np->n == n)
242                                 break ;
243                           np = np->nextnp ;
244                     }
245                     if(np == NULL) {
246                           printf(" Non-existent node %d referred to in
247   element %d\n",n,elmntp->n) ;
248                           rc = -1 ;
249                     }
250                     elmntp->nps[j] = np ;
251               }
252               else
```

```
253                         elmntp->nps[j] = NULL ;
254         }
255         for(j=0;j<N.params;j++)
256                 elmntp->p[j] = get_dbl(f) ;
257         elmntp->matrices = NULL ;
258         if(i < N.elms - 1)
259                 elmntp->nextelp = elmntp + 1 ;
260         else
261                 elmntp->nextelp = NULL ;
262         return(rc) ;
263 }
264
265 int     get_belm(f,i)
266 FILE    *f ;
267
268 {
269         int             j, jj, nnds, n, rc ;
270         struct belement       *belmntp ;
271         struct node    *np;
272
273         belmntp = gp.B + i ;
274         rc = 0 ;
275         belmntp->n = get_int(f) ;
276         belmntp->vtype = get_int(f);
277         belmntp->gtype = get_int(f);
278         belmntp->nnds =  nsf(belmntp->vtype) ;
279         nnds = nsf(belmntp->gtype);
280         if(nnds > belmntp->nnds) belmntp->nnds = nnds ;
281         for(j=0;j<belmntp->nnds;j++) {
282                 n = get_int(f) ;
283                 np = gp.N ;
284                 for(jj=0;jj<N.nodes;jj++) {
285                         if(np->n == n)
286                                 break ;
287                         np = np->nextnp ;
288                 }
289                 if(np == NULL) {
290                         printf(" Non-existent node %d referred to in element
291 %d\n",n,belmntp->n) ;
292                         rc = -1 ;
293                 }
294                 belmntp->nps[j] = np ;
295         }
296         for(j=0;j<N.bparams;j++)
297                 belmntp->p[j] = get_dbl(f) ;
298         for(j=0;j<N.vars;j++) {
299                 belmntp->bcs[j] = get_int(f) ;
300         }
301         if(i < N.belms - 1)
302                 belmntp->nextbelp = belmntp + 1;
303         else
304                 belmntp->nextbelp = NULL ;
305         return(rc) ;
306 }
307
308 double          get_dbl(fptr)
309 FILE    *fptr ;
310
311 {
312         char    s[25] ;
313         int     n ;
314         double atof();
315
```

```
316         n  = getnumstr(fptr,s) ;
317         return( atof(s) ) ;
318   }
319
320   int     get_int(fptr)
321   FILE    *fptr ;
322
323   {
324         char    s[25] ;
325         int     n ;
326         double        atof(), d;
327
328         n = getnumstr(fptr,s) ;
329         d = atof(s) ;
330         return( (int)  d ) ;
331   }
332
333   int     getnumstr(fptr,s)
334   char    *s ;
335   FILE    *fptr ;
336
337   {
338         register int  c ;
339         register char *cs ;
340
341         cs = s ;
342         while( (c = getc(fptr)) != EOF) {
343               if((isdigit(c) != 0) || (c == '.') || (c == '-') || (c == '+')) {
344                     *cs++ = c ;
345                     while( (c = getc(fptr)) != EOF) {
346                           if((isdigit(c) != 0) || (c == '.') || (c == 'e') ||
347   (c == 'E') || (c == '-') || (c == '+') )
348                                 *cs++ = c ;
349                           else
350                                 break ;
351                     }
352                     *cs = '\0' ;
353                     break ;
354               }
355         }
356         return(cs - s) ;
357   }
358
359   int           set_bvalues()
360
361   {
362         int           i, j ;
363         struct node   *nptr ;
364
365         j = 0 ;
366         nptr = gp.N ;
367         for(i=0;i<N.nodes;i++) {
368               nptr->i = i ;
369               gp.iptrs[i] = nptr ;
370               nptr = nptr->nextnp ;
371         }
372         N.ukns = N.nodes * N.vars ;
373   /*    printf(" Number of unknowns = %d\n",N.ukns) ;          */
374         return(j) ;
375   }
376
377   int           output(nin)
378   int           nin ;
```

```
379
380   {
381         int           err,i,fer ;
382         FILE          *fptr, *put_fptr() ;
383         struct node   *np ;
384         struct element       *elp ;
385         struct belement *belp ;
386
387         if((fptr = put_fptr(nin) ) != NULL) {
388               if( (err = put_control(fptr)) == 0) {
389                     fprintf(fptr,"\n Node Information \n");
390                     fprintf(fptr,"\n Node #, Coordinates, Parameters,
391   Variables\n\n");
392                     np = gp.N ;
393                     for(i=0;i<N.nodes;i++) {
394                           put_node(fptr,i,np) ;
395                           np = np->nextnp ;
396                     }
397                     fprintf(fptr,"\n Element Information \n");
398                     fprintf(fptr,"\n Element #, vtype, gtype, nodes\n\n");
399                     elp = gp.El ;
400                     for(i=0;i<N.elms;i++) {
401                           put_elm(fptr,i,elp) ;
402                           elp = elp->nextelp ;
403                     }
404                     fprintf(fptr,"\n Boundary Element #, vtype, gtype, nodes,
405   boundary condition codes\n\n");
406                     belp = gp.B ;
407                     for(i=0;i<N.belms;i++) {
408                           put_belm(fptr,i,belp) ;
409                           belp = belp->nextbelp ;
410                     }
411               }
412               else
413                     printf(" Control Variable Error %d\n",err) ;
414         }
415         else
416               printf(" Unable to open that file\n") ;
417
418         if((fer = ferror(fptr))!= NULL || (fer != 0) ){
419               printf(" Write error in routine Output \n Error ");
420               clearerr(fptr) ;
421               if(fptr != stdin) fclose(fptr) ;
422               return(fer) ;
423         }
424         else {
425               if(fptr != stdin) fclose(fptr) ;
426               return(err);
427         }
428   }
429
430   int    put_control(f)
431   FILE   *f ;
432
433   {
434         int    i, ii ;
435
436         fprintf(f," Transient analysis = %d\n",N.trans) ;
437         if(N.trans == 1){
438               fprintf(f," Number of Time Steps = %d\n",tvals.nsteps) ;
439               fprintf(f," Delta t Acceleration Factor = %5.3f\n",tvals.dtfac) ;
440               fprintf(f," Time = %7.5f\n",tvals.t) ;
441               fprintf(f," Delta t = %7.5f\n", tvals.dt) ;
```

```
442                 fprintf(f," Theta = %5.3f\n", tvals.theta) ;
443         }
444         fprintf(f," Dimensions = %d\n",N.dims) ;
445         fprintf(f," Number of Variables = %d\n",N.vars) ;
446         fprintf(f," [K] governing equation numbers \n") ;
447                 for(i=0;i<N.vars;i++) {
448                         for(ii=0;ii<N.vars;ii++) {
449                                 fprintf(f,"\t %d ",N.Keqns[i*(N.vars+1) + ii]) ;
450                         }
451                         fprintf(f,"\t\t %d \n",N.Keqns[i*(N.vars+1) + N.vars]) ;
452                 }
453         fprintf(f," Number of Parameters = %d\n",N.params) ;
454         fprintf(f," Number of Boundary Parameters = %d\n",Mesh_code*N.bparams) ;
455         fprintf(f," Number of Nodes = %d\n",N.nodes) ;
456         fprintf(f," Number of Elements = %d\n",N.elms) ;
457         fprintf(f," Number of Boundary Elements = %d\n",N.belms) ;
458         if(Mesh_code == -1){
459                 fprintf(f," Number of Element in X direction = %d\n",Mesh.nx) ;
460                 fprintf(f," Number of Blocks in X direction = %d\n",Mesh.nbx) ;
461                 fprintf(f," Number of Element in Y direction = %d\n",Mesh.ny) ;
462                 fprintf(f," Number of Blocks in Y direction = %d\n",Mesh.nby) ;
463         }
464         fprintf(f, " \n") ;
465
466         return(0) ;
467 }
468
469 int             put_node(f,i,nodep)
470 FILE            *f ;
471 int             i ;
472 struct node     *nodep ;
473
474 {
475         int     j ;
476
477         put_int(f,nodep->n)  ;
478         for(j=0;j<N.dims;j++)
479                 put_dbl(f,(double) nodep->x[j]) ;
480         for(j=0;j<N.params;j++)
481                 put_dbl(f,(double) nodep->p[j]) ;
482         for(j=0;j<N.vars;j++)
483                 put_dbl(f,(double) nodep->u[j]) ;
484         fprintf(f," \n");
485         return(0) ;
486 }
487
488 int             put_elm(f,i,elmntp)
489 int             i ;
490 FILE            *f ;
491 struct element          *elmntp ;
492
493 {
494         int             j;
495
496         put_int(f,elmntp->n) ;
497         put_int(f,elmntp->vtype);
498         put_int(f,elmntp->gtype);
499         for(j=0;j<elmntp->nnds;j++) {
500                 if(elmntp->nps[j] != NULL)
501                         put_int(f,(elmntp->nps[j])->n) ;
502                 else
503                         put_int(f,-1) ;
504         }
```

```
505         for(j=0;j<N.params;j++)
506                 put_dbl(f,(double) elmntp->p[j]) ;
507         fprintf(f," \n");
508         return(0) ;
509 }
510
511 int             put_belm(f,i,belmntp)
512 int             i ;
513 FILE            *f ;
514 struct belement *belmntp ;
515
516 {
517         int             j;
518
519         put_int(f,belmntp->n) ;
520         put_int(f,belmntp->vtype);
521         put_int(f,belmntp->gtype);
522         for(j=0;j<belmntp->nnds;j++)
523                 put_int(f,(belmntp->nps[j])->n) ;
524         for(j=0;j<N.bparams;j++)
525                 put_dbl(f,(double) belmntp->p[j]) ;
526         for(j=0;j<N.vars;j++)
527                 put_int(f,belmntp->bcs[j]) ;
528         fprintf(f," \n");
529         return(0) ;
530 }
531
532 int     put_dbl(fptr,d)
533 FILE    *fptr ;
534 double d;
535
536 {
537         fprintf(fptr,"\t%14.5g",d) ;
538         return(0) ;
539 }
540
541 int     put_int(fptr,i)
542 FILE    *fptr ;
543 int     i ;
544
545 {
546         fprintf(fptr,"\t%d",i) ;
547         return(0) ;
548 }
```

fecore.c

```
1   #include "fe.h"
2
3   #define KE(A,B)         ( *(ep.K + ep.n*A + B) )
4   #define SE(A,B)         ( *(ep.S + ep.n*A + B) )
5   #define ME(A,B)         ( *(ep.M + ep.n*A + B) )
6   #define FE(B)                   ( *(ep.F + B) )
7   #define Min_mem         20000
8
9   struct control                  N ;
10  struct pointers         gp;
11  struct RegMesh                  Mesh ;
12  struct fxnodes                  *first_fnp, *next_fnp ;
13  struct transient        tvals ;
14  int                     Nukns, Nrfixed,  Fac, *BFnodes ;
15  struct elmpointers      ep ;
16  struct eqnset                   eqnsets[4] ;
17  int                     BandWidth , Max_iter, nx_s = 0, steady_code;
18  double                          uchange ;
19  float                   x_s[10] ;
20
21  extern double           defndp , defelp ;
22
23  int             init_gp()
24
25  {
26          int     i ;
27
28          gp.N = NULL ;
29          gp.El = NULL ;
30          gp.B = NULL ;
31          gp.S = NULL ;
32          gp.M = NULL ;
33          for(i=0;i<4;i++) {
34                  eqnsets[i].Kp = NULL ;
35                  eqnsets[i].Fp = NULL ;
36                  eqnsets[i].diagp = NULL ;
37                  eqnsets[i].neqns = NULL ;
38          }
39          ep.K = NULL ;
40          ep.F = NULL ;
41          ep.S = NULL ;
42          ep.M = NULL ;
43          first_fnp = NULL ;
44          Mesh.nx = 10 ;
45          Mesh.ny = 10 ;
46          Mesh.nbx = 1 ;
47          Mesh.nby = 1 ;
48          Mesh.eltype = 211 ;
49          Mesh.maptype = 211 ;
50          Mesh.firstmap = (struct blockmap *) calloc(24,sizeof(struct blockmap)) ;
51          Mesh.firstmap->xnodes[0] = 100. ; Mesh.firstmap->ynodes[0] = 0.0 ;
52          Mesh.firstmap->xnodes[1] = 100. ; Mesh.firstmap->ynodes[1] = 100. ;
53          Mesh.firstmap->xnodes[2] = 0.0 ; Mesh.firstmap->ynodes[2] = 100. ;
54          Mesh.firstmap->xnodes[3] = 0.0 ; Mesh.firstmap->ynodes[3] = 0.0 ;
55          tvals.nsteps = 10 ;
56          tvals.t = 0.0 ;
57          tvals.dt = 10. ;
58          tvals.theta = 0.5 ;
59          tvals.dtfac = 1.0 ;
60          N.sym = 1 ;
61  }
62
63  int             MkLapMesh(theMesh)
```

```
64  struct RegMesh *theMesh ;
65
66  {
67          N.trans = theMesh->con.trans = 1 ;
68          N.dims = theMesh->con.dims = 2 ;
69          N.vars = theMesh->con.vars = 1 ;
70          N.Keqns[0] = theMesh->con.Keqns[0] = 1 ;
71          N.Keqns[1] = theMesh->con.Keqns[1] = 1 ;
72          N.params = theMesh->con.params = 8 ;
73          N.bparams = theMesh->con.bparams = 2 ;
74          defndp = 0.00 ;
75          defelp = 0.00 ;
76          MakeMesh(theMesh) ;
77          return(0) ;
78  }
79
80  int             steady_state(nvar, elcode)
81  int             nvar, elcode ;
82  {
83          int     i ;
84          char    t = 'n';
85
86          N.trans = 1 ;
87          tvals.dt = 1000000000000000. ;
88          tvals.t += tvals.dt ;
89          tvals.theta = 1.0 ;
90          tvals.dtfac = 100. ;
91          BandWidth = 1 ;
92          steady_code = 1 ;
93
94          printf(" Do you want to use 'banded storage'(faster)?\n") ;
95          scanf(" %c",&t) ;
96          if(t == 'y'){
97                  printf(" Input the Bandwidth of the Mesh ?\n  ->") ;
98                  scanf(" %d",&BandWidth) ;
99                  printf(" %d\n",BandWidth) ;
100         }
101         else
102                 printf(" Using skylined storage(slow!)\n") ;
103
104         printf(" Now Solving\n") ;
105
106         assemble(nvar,0,0,0) ;
107         solve(nvar,1,0) ;
108         printf(" Solved First step\n") ;
109         tvals.t += tvals.dt ;
110         assemble(nvar,0,0,0) ;
111         solve(nvar,1,0) ;
112         printf(" Done \n") ;
113
114         return(0) ;
115 }
116
117 int             transient(nvar,elcode)
118 int             nvar,elcode ;
119
120 {
121         int             int_code = 0 ,i, j, nin, k, n_code = -1, in, t_int;
122         struct node     *np ;
123         struct element          *elp ;
124         FILE            *f, *f2, *put_fptr() ;
125         char            t ;
126         float           x, y ;
```

```
127         double        x_p ;
128
129         BandWidth = 1 ;
130         steady_code = 0 ;
131         N.trans = 1 ;
132         nin = 1 ;
133
134         printf(" Do you want to use 'banded storage'(faster)?\n  ->") ;
135         scanf(" %c",&t) ;
136         if(t == 'y'){
137               printf(" Input the Bandwidth of the Mesh ?\n  ->") ;
138               scanf(" %d",&BandWidth) ;
139               printf(" %d\n",BandWidth) ;
140         }
141         else
142               printf(" Using skylined storage(slow!)\n") ;
143
144
145         t = 'n' ;
146
147         if(tvals.theta == 0.5){
148               printf(" Do you want to set up a Slug as an initial condition ?\n
149   ->") ;
150               scanf(" %c",&t) ;
151               if(t == 'y')
152                     set_slug() ;
153               t = 'n' ;
154               printf(" Do you want output at certain times (max 5)?\n  ->") ;
155               scanf(" %c",&t) ;
156               if(t == 'y'){
157                     int_code = 1 ;
158                     printf(" Input the interval between output?\n  ->") ;
159                     scanf(" %d",&in) ;
160                     printf(" Output will be generated every %d steps\n",in) ;
161               }
162
163               printf(" Do you want a time history of the results ?\n  ->") ;
164               scanf(" %c",&t) ;
165         }
166         if(t == 'y') {
167               f = put_fptr(1) ;
168               printf(" Input the interval at which to output?\n  ->") ;
169               scanf(" %d",t_int) ;
170               printf(" Input the number of x stations where you want the
171   results (max 10)?\n  ->") ;
172               scanf(" %d",&nx_s) ;
173               printf(" Input the x_coordinates\n");
174               for(k=0;k<nx_s;k++){
175                     printf(" Input %d point?\n  ->",k) ;
176                     scanf(" %f",&(x_s[k])) ;
177                     printf(" Output will be generated at x = %f\n",x_s[k]) ;
178               }
179               fprintf(f," Delta t =\t %7.5f\t", tvals.dt) ;
180               fprintf(f," Theta =\t %5.3f", tvals.theta) ;
181               fprintf(f," Time =\t %7.5f",tvals.t) ;
182               fprintf(f,"\nNode # \t\t\t") ;
183
184               for(k=0;k<nx_s;k++){
185                     np = gp.N ;
186                     for(j=0;j<N.nodes;j++){
187                           if( np->x[0] == x_s[k] ){
188                                 fprintf(f,"\t%d",np->n) ;
189                           }
```

```
190                             np = np->nextnp ;
191                     }
192             }
193
194             fprintf(f,"\nX \t\t\t") ;
195             for(k=0;k<nx_s;k++){
196                     np = gp.N ;
197                     for(j=0;j<N.nodes;j++){
198                             if(  np->x[0] == x_s[k] ){
199                                     fprintf(f,"\t%f",np->x[0]) ;
200                             }
201                             np = np->nextnp ;
202                     }
203             }
204             fprintf(f,"\nY \t\t\t") ;
205             for(k=0;k<nx_s;k++){
206                     np = gp.N ;
207                     for(j=0;j<N.nodes;j++){
208                             if(  np->x[0] == x_s[k] ){
209                                     fprintf(f,"\t%f",np->x[1]) ;
210                             }
211                             np = np->nextnp ;
212                     }
213             }
214             fprintf(f,"\nC init \t\t\t") ;
215             for(k=0;k<nx_s;k++){
216                     np = gp.N ;
217                     for(j=0;j<N.nodes;j++){
218                             if(  np->x[0] == x_s[k] ){
219                                     fprintf(f,"\t%f",np->u[0]) ;
220                             }
221                             np = np->nextnp ;
222                     }
223             }
224     }
225
226
227     Mesh.eltype = (gp.El)->vtype ;
228
229     printf("\n Now Solving !\n") ;
230     for(i=0;i<tvals.nsteps;i++) {
231
232             tvals.t += tvals.dt ;
233             assemble(nvar,0,0,0) ;
234             solve(nvar,1,0) ;
235
236             k = (int) (tvals.t/tvals.dt) ;
237             if(int_code == 1 && nin < 7){
238                     if((i+1) % in == 0){
239                             nin += 1 ;
240                             output(nin) ;
241                     }
242             }
243
244             if(t == 'y'  && i > 1 ){
245                     if ( i % t_int == 0)
246                             put_results(f) ;
247             }
248
249             printf(" Time = %lf Finished %d of %d time steps \t Δu =
250 %g\n",tvals.t,i+1, tvals.nsteps, uchange ) ;
251     }
252     if(t == 'y')
```

```
253                 fclose(f) ;
254
255         return(0) ;
256 }
257
258
259 int     set_slug()
260
261 {
262         int             j ;
263         float           x_loc, y_loc, sigma_x, sigma_y, mag ;
264         struct node     *np ;
265
266         printf(" Input the x and y coordinates for the location of the Slug
267 Center?\n  ->") ;
268         scanf(" %f %f",&x_loc,&y_loc) ;
269         printf(" Input the sigma_x and sigma_y of the distribution?\n  ->") ;
270         scanf(" %f %f",&sigma_x,&sigma_y) ;
271         printf(" Input the magnitude of the Maximum value @ Center of the
272 distribution?\n  ->") ;
273         scanf(" %f",&mag) ;
274
275         np = gp.N ;
276         for(j=0;j<N.nodes;j++){
277                 np->u[0] =mag * exp(-( (np->x[0]-x_loc)*(np->x[0]-
278 x_loc)/(sigma_x*sigma_x)
279                                         + (np->x[1]-y_loc)*(np->x[1]-
280 y_loc)/(sigma_y*sigma_y)  )/2.) ;
281                 if(np->u[0] < 0.00000000001)
282                         np->u[0] = 0 ;
283
284                 np = np->nextnp ;
285         }
286         printf(" Updated the Mesh !\n") ;
287         return(0) ;
288 }
289
290 int put_results(f)
291 FILE *f ;
292 {
293         int             j,k ;
294         struct node     *np ;
295         float           x , y  ;
296
297         fprintf(f,"\n") ;
298         fprintf(f," Time =\t%7.5f\t ",tvals.t) ;
299         fprintf(f," C\t") ;
300         for(k=0;k<nx_s;k++){
301                 np = gp.N ;
302                 for(j=0;j<N.nodes;j++){
303                         if(  np->x[0] == x_s[k] ){
304                                 fprintf(f,"\t%f",np->u[0]) ;
305                         }
306                         np = np->nextnp ;
307                 }
308         }
309         return(0) ;
310 }
311
312
313
314 int             update_PGKe(gbfp,gtfp,nbf,ntf,np,vcode)
315 int             nbf, ntf, vcode ;
```

```
316   struct node   *np[] ;
317   struct govfuncs      *gbfp, *gtfp ;
318
319   {
320         int                           err, n ;
321         register int                  i, j, nk ;
322         register struct govfuncs      *f, *v ;
323         register double                     *Kep, *Sep, *Fep ;
324         double                              diff, conv;
325
326         if(vcode<0)
327               n = N.vars ;
328         else
329               n = 1 ;
330
331         nk = ep.n ;
332         f = gbfp ;
333         v = gtfp ;
334         Kep = ep.K ;
335         Sep = ep.S ;
336         Fep = ep.F ;
337
338         i = ntf  ;
339         while(--i >= 0) {
340               j = nbf ;
341               while(--j >= 0) {
342
343
344                     *(Kep + nk*i + j) += (v->f[i] * (f->p[0] * f->dfdx[j] + f-
345   >p[1] * f->dfdy[j])
346   /* weak statement */                  + v->dfdx[i] * (f->p[2] * f->dfdx[j] + f-
347   >p[3] * f->dfdy[j])
348                                         + v->dfdy[i] * (f->p[4] * f->dfdx[j] + f-
349   >p[5] * f->dfdy[j]) )* f->detJ;
350
351                     if(N.trans > 0)
352                           *(Sep + nk*i + j) += (v->f[i] * f->f[j]) * f->detJ
353   ;
354               }
355
356               *(ep.F + i) += ( f->p[6] * v->f[i] ) * f->detJ ;
357
358         }
359         return(0) ;
360   }
361
362   int                  update_BKe(elmntp,gbf,gtf,nbf,ntf,np,nvar,dS)
363   int                  nbf, ntf ;
364   double               dS ;
365   struct node          *np[] ;
366   struct belement      *elmntp ;
367   struct shapefuncs    *gbf, *gtf ;
368
369   {
370         int    i, j, n ;
371         double BoundK(), BoundF() ;
372
373         n = N.vars ;
374         for(i=0;i<ntf;i++) {
375               for(j=0;j<nbf;j++) {
376                     KE(i*n,j*n) += BoundK(elmntp,i,j,gbf,gtf,dS) ;
377               }
378               FE(i*n) += BoundF(elmntp,i,gtf,dS) ;
```

```
379         }
380 }
381
382 double                  BoundK(belp,i,j,fgp,ftp,dS)
383 int                     i,j ;
384 double                  dS ;
385 struct shapefuncs       *fgp, *ftp ;
386 struct belement         *belp ;
387
388 {
389         if(belp->gtype == 1)
390                 return(belp->p[0]) ;
391         return(belp->p[0]* fgp->f[i] * ftp->f[j] * dS ) ;
392 }
393
394 double                  BoundF(belp,i,fgp,dS)
395 int                     i ;
396 double                  dS ;
397 struct shapefuncs       *fgp ;
398 struct belement         *belp ;
399
400 {
401         if(belp->gtype == 1)
402                 retúrn( -belp->p[1]) ;
403         return( -belp->p[1] * fgp->f[i] * dS ) ;
404 }
405
406
407
408 int             assemble(varnum,code,nset,elcode)
409 int             varnum, code, nset, elcode ;
410
411 {
412         double                  *K_init(), *L_init(),*F_init() ;
413         struct fxnodes                  *fn_init();
414         struct element                  *elp, tel ;
415         struct belement         *belp, tbel ;
416         double                  *p ;
417         int                     i,j,ns,ntf,istart,iend,ind  ;
418         long                    lj ;
419         char                    m1[255], m2[255], m3[255], m4[255] ;
420
421
422         istart = 0 ;
423         iend = N.elms ;
424         if(N.trans == 0)
425                 tvals.dt = 1.0 ;
426         uchange = 0.0 ;
427         if(code< 2) {
428                 if(eqnsets[nset].Kp != NULL)
429                         free(eqnsets[nset].Kp) ;
430                 if( (eqnsets[nset].Kp = K_init(varnum,&eqnsets[nset].diagp,code))
431 == NULL )
432                         return(-1) ;
433
434
435                 if(N.sym != 0) {
436                         if(eqnsets[nset].Lp != NULL )
437                           free(eqnsets[nset].Lp) ;
438                         if( (eqnsets[nset].Lp =
439 L_init(varnum,&eqnsets[nset].diagp,code)) == NULL )
440                                 return(-1) ;
441                 }
```

```
442
443                 eqnsets[nset].neqns = Nukns ;
444                 if(steady_code == 1){
445                         sprintf(m1," Assembling K matrix and F vector ");
446                         sprintf(m2," Number of equations to be solved = %d
447 ",Nukns) ;
448                         sprintf(m3," Elements in K matrix - %ld
449 ",*(eqnsets[nset].diagp+Nukns-1)+1) ;
450                         sprintf(m4," ") ;
451                         StartMes(m1,m2,m3,m4) ;
452                 }
453
454                 if(eqnsets[nset].Fp != NULL) {
455                         free(eqnsets[nset].Fp) ;
456                 }
457                 if( ( eqnsets[nset].Fp = F_init(Nukns) )  == NULL  )
458                         return(-1) ;
459         }
460         else if (code == 2) {
461                 ind = 0 ;
462                 for(i=0;i<N.nodes;i++) {
463                         gp.iptrs[i]->ui = ind ;
464                         ind++ ;
465                 }
466         }
467
468         if(code < 1) {
469
470                 if( (ep.n = ep_init() ) == 0 )
471                         return(-1) ;
472         }
473         for(i=0;i<eqnsets[nset].neqns;i++)
474                 *(eqnsets[nset].Fp + i) = 0.0 ;
475
476         elp = gp.El ;
477         for(i=istart;i<iend;i++) {
478                 if( i % 10 == 0 && steady_code == 1 ) {
479                         sprintf(m4," Assembling Element # %d of %d ",i,N.elms) ;
480                         UpDateMes(m1,m2,m3,m4) ;
481                 }
482
483                 ns = get_PGKe(elp,&tel,varnum,elcode,&ntf) ;
484
485
486         put_Ke(ns,ntf,&tel,varnum,eqnsets[nset].Kp,eqnsets[nset].Lp,eqnsets[nset
487 ].Fp,eqnsets[nset].diagp,code) ;
488                 elp = elp->nextelp ;
489         }
490         if( steady_code == 1){
491                 sprintf(m4," Assembling Elements done !");
492                 UpDateMes(m1,m2,m3,m4);
493         }
494
495         belp = gp.B ;
496         for(i=0;i<N.belms;i++) {
497                 ns = get_bKe(belp,&tbel,varnum,elcode,&ntf) ;
498                 if(ns > 0 )
499
500         put_Ke(ns,ntf,belp,varnum,eqnsets[nset].Kp,eqnsets[nset].Lp,eqnsets[nset
501 ].Fp,eqnsets[nset].diagp,code) ;
502                 belp = belp->nextbelp ;
503         }
504         if(steady_code == 1){
```

```
505                 sprintf(m4," Assembling Boundary Elements done !");
506                 UpDateMes(m1,m2,m3,m4);
507         }
508
509         if(code < 2)
510                 EndMes() ;
511
512         return(0) ;
513 }
514
515
516 int             ep_init()
517
518 {
519         int             i, ns, nsm ;
520         struct element      *elp ;
521
522         if(ep.K != NULL)
523                 free(ep.K) ;
524         elp = gp.El ;
525         nsm = 0 ;
526
527         for(i=0;i<N.elms;i++) {
528                 nsm = ( (ns = nsf(elp->vtype)) > nsm ) ? ns : nsm ;
529                 elp = elp->nextelp ;
530         }
531         nsm *= N.vars ;
532         if( (ep.K = (double *) calloc(nsm*nsm, sizeof(double)) ) == NULL )
533                 return(0) ;
534         if(ep.F != NULL)
535                 free(ep.F) ;
536         if( (ep.F = (double *) calloc(nsm, sizeof(double)) ) == NULL )
537                 return(0) ;
538         if(N.trans > 0) {
539                 if(ep.S != NULL)
540                         free(ep.S) ;
541                 if( (ep.S = (double *) calloc(nsm*nsm, sizeof(double)) ) == NULL
542 )
543                         return(0) ;
544         }
545         if(N.trans > 1) {
546                 if(ep.M != NULL)
547                         free(ep.M) ;
548                 if( (ep.M = (double *) calloc(nsm*nsm, sizeof(double)) ) == NULL
549 )
550                         return(0) ;
551         }
552         return(nsm) ;
553 }
554
555
556 double          *K_init(nvar,thediagp,code)
557 int             nvar, code ;
558 long            *(*thediagp) ;
559
560 {
561         int             nj, i, j, nf , ind;
562         double              *kp, *dp ;
563         unsigned        elsize ;
564         long            bigint, nelem,*diagp, ir ;
565
566         nelem = 0 ;
567         for(i=0;i<N.nodes;i++) {
```

```
568                 gp.iptrs[i]->ui = 9999 ;
569                 nelem++ ;
570         }
571         Nukns = nelem ;
572         if(*thediagp != NULL)
573                 free(*thediagp) ;
574         if((*thediagp = (long int *) calloc((int)nelem,sizeof(long))) == NULL)
575                 return(NULL) ;
576         diagp = *thediagp ;
577         *diagp = 0 ;
578         ind = 0 ;
579         diagp++ ;
580         gp.iptrs[0]->ui = ind ;
581         ind++ ;
582         for(i=1;i<N.nodes;i++) {
583                 gp.iptrs[i]->ui = ind ;
584                 ind++ ;
585                 if(code == 0){
586                         if( BandWidth == 1)
587                                 nf =  (mpd_i(i,nvar)  + 1) * N.vars ;
588                         else {
589                                 if(ind < BandWidth)
590                                         nf = ind ;
591                                 else
592                                         nf = BandWidth;
593                         }
594
595                 }
596                 else if(code == -1)
597                         nf = 1 ;
598                 else if(code == 2) {
599                         if(ind == 1)
600                                 nf = 1 ;
601                         else if(ind == 2)
602                                 nf = 2 ;
603                         else
604                                 nf = 3 ;
605                 }
606                 else
607                         nf = 2 ;
608                 if(diagp != *thediagp)
609                         *diagp = *(diagp-1) + nf ;
610                 diagp++ ;
611         }
612         nelem = *(diagp-1) + 1 ;
613         elsize = sizeof(double) ;
614         bigint =  nelem * (long) elsize  ;
615         if((kp =  dp = (double *) lmalloc(bigint))==NULL){          /* NON
616 STANDARD FUNCTION    */                  /* NON STANDARD FUNCTION CALL */
617                 printf(" Insufficent Memory avaliable") ;
618                 exit(0) ;
619         }
620         for(il=0;il<nelem;il++)
621                 *(dp++) = 0.0 ;
622         return(kp) ;
623 }
624
625
626 int             mpd_i(ind,nvar)
627 int             ind , nvar;
628
629 {
630         int                             i,j,n,mpd,td ;
```

```
631         register struct element     *elmntp ;
632
633         mpd = 0 ;
634
635         if(N.dims == 1)
636                 return(3) ;
637         elmntp = gp.El ;
638         for(i=0;i<N.elms;i++) {
639                 n = -1 ;
640                 for(j=0;j<elmntp->nnds;j++) {
641                         if(elmntp->nps[j] != NULL) {
642                                 if(gp.iptrs[ind]->n == (elmntp->nps[j])->n )
643                                         n = gp.iptrs[ind]->n ;
644                         }
645                 }
646                 if( n >= 0 ) {
647                         for(j=0;j<elmntp->nnds;j++) {
648                                 if(elmntp->nps[j] != NULL) {
649                                         if((elmntp->nps[j])->n >= 0)
650                                                 td = d_i(ind,(elmntp->nps[j])->i) ;
651                                         mpd = (mpd > td ) ? mpd : td ;
652                                 }
653                         }
654                 }
655                 elmntp = elmntp->nextelp ;
656         }
657         return(mpd) ;
658 }
659
660 int             d_i(n1,n2)
661 int             n1, n2 ;
662
663 {
664         return(n1 - n2) ;
665 }
666
667
668 double *L_init(nvar,thediagp,code)
669 int     nvar,code ;
670 long    *(*thediagp) ;
671
672 {
673         int             nj, i, j, nf , ind;
674         double                  *kp, *dp ;
675         unsigned        elsize ;
676         long            bigint, nelem, il, *diagp ;
677
678         diagp = *thediagp ;
679         for(i=0;i<N.nodes;i++) {
680                 if(isvarfixed(i,nvar) == 0) {
681                         diagp++ ;
682                 }
683         }
684
685         nelem = *(diagp-1) + 1 ;
686         elsize = sizeof(double) ;
687         bigint =  nelem * (long) elsize  ;
688         if((kp =  dp = (double *) lmalloc(bigint))==NULL){
689 /* NON STANDARD FUNCTION CALL */
690                 printf(" Insufficent Memory avaliable") ;
691                 exit(0) ;
692         }
693         for(il=0;il<nelem;il++)
```

```
694                 *(dp++) = 0.0 ;
695         return(kp) ;
696 }
697
698
699 double *F_init(num)
700 int             num ;
701
702 {
703         double          *p ;
704         unsigned        nelem, elsize ;
705
706         nelem = num ;
707         elsize = sizeof(double) ;
708         p = (double *) calloc(nelem,elsize) ;
709         return(p) ;
710 }
711
712
713
714 int             put_Ke(ns,nt,elmntp,nvar,Kp,Lp,Fp,diagp,code)
715 int             ns, code ;
716 double          *Kp,  *Lp, *Fp ;
717 struct element      *elmntp ;
718 long            *diagp ;
719
720 {
721         int             i, in, j, ii, jj, jn, frow, fcol, row, col, tj, isv ;
722         struct node     *inp, *jnp ;
723         struct fxnodes      *the_fnp ;
724
725         for(i=0;i<nt;i++) {
726                 inp = elmntp->nps[i] ;
727                 frow = inp->i * N.vars ;
728                 for(j=0;j<ns;j++) {
729                         jnp = elmntp->nps[j] ;
730                         fcol = jnp->i * N.vars;
731                         for(ii=0;ii<N.vars;ii++) {
732                                 row = frow + ii ;
733                                 for(jj=0;jj<N.vars;jj++) {
734                                         col = fcol + jj ;
735                                         if( (col >= row) && ( code < 2 ) && ( code >
736 -1 ) ) {
737                                                 if(N.trans == 0)
738                                                         *(Kp + *(diagp+col) - col +
739 row ) += KE(i*N.vars+ii,j*N.vars+jj) ;
740                                                 else
741                                                         *(Kp + *(diagp+col) - col +
742 row ) += SE(i*N.vars+ii,j*N.vars+jj) ;
743                                         }
744                                         if( (col <= row) && (N.sym != 0 ) ) {
745                                                 if(N.trans == 0)
746                                                         *(Lp + *(diagp+row) - row +
747 col ) += KE(i*N.vars+ii,j*N.vars+jj) ;
748                                                 else
749                                                         *(Lp + *(diagp+row) - row +
750 col ) += SE(i*N.vars+ii,j*N.vars+jj) ;
751                                         }
752                                         if( (col == row) && ( code == -1 ) ) {
753                                                 if(N.trans == 0)
754                                                         *(Kp + row ) +=
755 KE(i*N.vars+ii,j*N.vars+jj) ;
756                                                 else
```

```
                                                        *(Kp + row ) +=
SE(i*N.vars+ii,j*N.vars+jj) ;
                                        }
                                        if(  (code == 2 ) || code == -1 ) {
                                                *(Fp  + row ) += -
KE(i*N.vars+ii,j*N.vars+jj) * jnp->u[jj] ;
                                        }
                                }
                        }
                        for(ii=0;ii<N.vars;ii++) {
                                row = frow + ii ;
                                *(Fp + row) += FE(i*N.vars+ii) ;
                        }
                }
        }

int             get_PGKe(elmntp,theElp,nvar,code,ntf)
struct element          *elmntp, *theElp ;
int             nvar, code, *ntf ;

{
        struct gausspts         g[MGPTS] ;
        struct shapefuncs       bf, tf, *fgp;
        struct node             *np[MNSF] ;
        struct govfuncs         gbf, gtf ;
        int                     i, j, nbf, ni, nm, dir ;
        double                  tdt, mtdt ;
        double                  *fptr ;

        initm(ep.K,ep.n*ep.n) ;
        if(N.trans > 0)
                initm(ep.S,ep.n*ep.n) ;
        initm(ep.F,ep.n) ;

        ElPick(elmntp,theElp,ntf,&dir) ;
        nbf = theElp->nnds ;

        if(code > 0 || elmntp->matrices == NULL) {
                ni = get_gspts(theElp->vtype,g) ;
                for(j=0;j<theElp->nnds;j++)
                        np[j] = theElp->nps[j];

                for(i=0;i<ni;i++) {
                        get_shape(theElp->vtype,&g[i],&bf) ;
                        get_shape(theElp->gtype,&g[i],&tf) ;
                        get_PGgf(theElp,&gbf,&gtf,&bf,&tf,np,g[i].w) ;
                        update_PGKe(&gbf,&gtf,nbf,*ntf,np,nvar) ;
                }
                if(code != 2 && (FreeMem() > Min_mem)) {
                        if(elmntp->matrices != NULL)
                                free(elmntp->matrices) ;
                        nm = *ntf  * (nbf + 1) ;
                        if(N.trans > 0)
                                nm += *ntf * nbf ;
                        if((elmntp->matrices = (double *)calloc(nm
,sizeof(double)) ) == NULL)
                                return(-1) ;
                        fptr = elmntp->matrices ;
                        for(i=0;i<*ntf;i++)
                                for(j=0;j<nbf;j++) {
                                        *fptr = KE(i,j) ;
```

```
820                                     fptr++ ;
821                             }
822                     for(i=0;i<*ntf;i++){
823                             *fptr = FE(i);
824                             fptr++ ;
825                     }
826                     if(N.trans > 0) {
827                             for(i=0;i<*ntf;i++) {
828                                     for(j=0;j<nbf;j++) {
829                                             *fptr = SE(i,j) ;
830                                             fptr++ ;
831                                     }
832                             }
833                     }
834             }
835         }
836         else {
837                 fptr = elmntp->matrices ;
838                 for(i=0;i<*ntf;i++)
839                         for(j=0;j<nbf;j++) {
840                                 KE(i,j) = *fptr ;
841                                 fptr++ ;
842                         }
843                 for(i=0;i<*ntf;i++) {
844                         FE(i) = *fptr ;
845                         fptr++ ;
846                 }
847                 if(N.trans > 0) {
848                         for(i=0;i<*ntf;i++) {
849                                 for(j=0;j<nbf;j++) {
850                                         SE(i,j) = *fptr ;
851                                         fptr++ ;
852                                 }
853                         }
854                 }
855         }
856         if(N.trans > 0) {
857                 tdt = tvals.theta * tvals.dt ;
858                 mtdt = tvals.dt * (1.0 - tvals.theta) ;
859                 for(i=0;i<*ntf;i++) {
860                         FE(i) *= tvals.dt ;
861                         for(j=0;j<nbf;j++) {
862                                 FE(i) += (SE(i,j) - mtdt * KE(i,j)) * (theElp-
863 >nps[j])->u[nvar] ;
864                                 SE(i,j) += tdt * KE(i,j) ;
865                         }
866                 }
867         }
868         return(nbf) ;
869 }
870
871 int             ElPick(elmntp,theElp,ntfp,dirp)
872 struct element          *elmntp, *theElp ;
873 int             *ntfp, *dirp ;
874
875 {
876         int     j;
877
878         *dirp = 1 ;
879
880         if(elmntp->vtype == 229)
881                 return(QUelPick(elmntp,theElp,ntfp,dirp)) ;
882         else if(elmntp->vtype == 239)
```

```
883                 return(CU2DPick(elmntp,theElp,ntfp,dirp)) ;
884         else if(elmntp->vtype == 233)
885                 return(CU_Stream_Pick(elmntp,theElp,ntfp,dirp)) ;
886
887
888
889         *ntfp = nsf(elmntp->gtype) ;
890         *theElp = *elmntp ;
891         return(0) ;
892 }
893
894 int                 get_PGgf(elp,gbfp,gtfp,bfp,tfp,np,w)
895 struct element              *elp ;
896 struct shapefuncs   *bfp, *tfp ;
897 struct node         *np[] ;
898 struct govfuncs     *gbfp,*gtfp ;
899 double              w ;
900
901 {
902         double J[NDIMS][NDIMS], get_J() ;
903         int    i, j ;
904
905         gbfp->dof = bfp->dof ;
906         gtfp->dof = tfp->dof ;
907
908         for(i=0;i<N.params;i++) {
909                 gbfp->p[i] = elp->p[i] ;
910                 for(j=0;j<gbfp->dof;j++)
911                         gbfp->p[i] += bfp->f[j] * np[j]->p[i] ;
912         }
913         for(j=0;j<bfp->dof;j++)
914                 gbfp->f[j] = bfp->f[j]      ;
915         for(j=0;j<tfp->dof;j++)
916                 gtfp->f[j] = tfp->f[j]      ;
917
918         gbfp->detJ = get_J(np,bfp,J) ;
919         gbfp->w = w ;
920         invertJ(J,gbfp->detJ) ;
921
922         for(j=0;j<bfp->dof;j++) {
923
924                 switch (N.dims) {
925
926                         case 1 :
927                                 gbfp->dfdx[j] = J[0][0] * bfp->dfdr[j] ;
928                                 gbfp->dfdy[j] = 0.0 ;
929                                 gbfp->dfdz[j] = 0.0 ;
930                                 break ;
931
932                         case 2 :
933                                 gbfp->dfdx[j] = J[0][0] * bfp->dfdr[j] + J[0][1] *
934 bfp->dfds[j] ;
935                                 gbfp->dfdy[j] = J[1][0] * bfp->dfdr[j] + J[1][1] *
936 bfp->dfds[j] ;
937                                 gbfp->dfdz[j] = 0.0 ;
938                                 break ;
939
940                         case 3 :
941                                 gbfp->dfdx[j] = J[0][0] * bfp->dfdr[j] + J[0][1] *
942 bfp->dfds[j] + J[0][2] *bfp->dfdt[j] ;
943                                 gbfp->dfdy[j] = J[1][0] * bfp->dfdr[j] + J[1][1] *
944 bfp->dfds[j] + J[1][2] *bfp->dfdt[j] ;
```

```
945                             gbfp->dfdz[j] = J[2][0] * bfp->dfdr[j] + J[2][1] *
946     bfp->dfds[j] + J[2][2] *bfp->dfdt[j] ;
947                             break ;
948                     }
949             }
950
951             for(j=0;j<tfp->dof;j++) {
952
953                     switch (N.dims) {
954
955                             case 1 :
956                                     gtfp->dfdx[j] = J[0][0] * tfp->dfdr[j] ;
957                                     gtfp->dfdy[j] = 0.0 ;
958                                     gtfp->dfdz[j] = 0.0 ;
959                                     break ;
960
961                             case 2 :
962                                     gtfp->dfdx[j] = J[0][0] * tfp->dfdr[j] + J[0][1] *
963     tfp->dfds[j] ;
964                                     gtfp->dfdy[j] = J[1][0] * tfp->dfdr[j] + J[1][1] *
965     tfp->dfds[j] ;
966                                     gtfp->dfdz[j] = 0.0 ;
967                                     break ;
968
969                             case 3 :
970                                     gtfp->dfdx[j] = J[0][0] * tfp->dfdr[j] + J[0][1] *
971     tfp->dfds[j] + J[0][2] *tfp->dfdt[j] ;
972                                     gtfp->dfdy[j] = J[1][0] * tfp->dfdr[j] + J[1][1] *
973     tfp->dfds[j] + J[1][2] *tfp->dfdt[j] ;
974                                     gtfp->dfdz[j] = J[2][0] * tfp->dfdr[j] + J[2][1] *
975     tfp->dfds[j] + J[2][2] *tfp->dfdt[j] ;
976                                     break ;
977                     }
978             }
979             gbfp->detJ *= gbfp->w ;
980             return(0) ;
981     }
982
```

```
1   #include "fe.h"
2
3   /*
4                   p[0] = u
5                   p[1] = v
6                   p[2] = Exx
7                   p[3] = Exy
8                   p[4] = Eyx
9                   p[5] = Eyy
10                  p[6] = P
11
12  */
13  extern struct transient tvals;
14  extern int              BandWidth ;
15
16  int             nodevalues(np)
17  struct node     *np ;
18
19  {
20          np->p[0] = 1.0 ;
21          np->p[1] = 0.0 ;
22          np->p[2] = 0.0 ;
23          np->p[3] = 0.0 ;
24          np->p[4] = 0.0 ;
25          np->p[5] = 0.0 ;
26          np->p[6] = 0.0 ;
27
28          return(0) ;
29  }
30
31  int             elvalues(elp)
32  struct element          *elp ;
33
34  {
35          return(0) ;
36  }
37
38  int             bottomb(belp)
39  struct belement *belp ;
40
41  {
42          belp->bcs[0] = 2 ;
43          belp->p[0] = 0.0 ;
44          belp->p[1] = 0.0 ;
45          return(0) ;
46  }
47
48  int             topb(belp)
49  struct belement *belp ;
50
51  {
52          belp->bcs[0] = 2 ;
53          belp->p[0] = 0.0 ;
54          belp->p[1] = 0.0 ;
55          return(0) ;
56  }
57
58  int             leftb(belp)
59  struct belement *belp ;
60
61  {
62          belp->bcs[0] = 3 ;
63          belp->p[0] = 0.0 ;
```

```
64          belp->p[1] = 0.0 ;
65          return(0) ;
66  }
67
68  int             rightb(belp)
69  struct belement *belp ;
70
71  {
72          belp->bcs[0] = 2 ;
73          belp->p[0] = 0.0 ;
74          belp->p[1] = 0.0 ;
75          return(0) ;
76  }
77
78
79  double          uexact(nvar,x,y,z,p)
80  int             nvar ;
81  double          x, y, z, p ;
82
83  {
84          double t ;
85
86          t = exp(-((x-6800.0)*(x-6800.0))/(2.*264.*264.)) ;
87          return(t) ;
88  }
89
90
91  int             update_p(np)
92  struct node     *np ;
93
94  {
95          return(0) ;
96  }
97
98
```

```
1   #include "fe.h"
2
3   #define       KE(A,B)         ( *(ep.K + ep.n*A + B) )
4   #define       SE(A,B)         ( *(ep.S + ep.n*A + B) )
5   #define       ME(A,B)         ( *(ep.M + ep.n*A + B) )
6   #define       FE(B)           ( *(ep.F + B) )
7
8   extern struct control             N ;
9   extern struct pointers            gp;
10  extern struct RegMesh             Mesh ;
11  extern struct fxnodes             *first_fnp ;
12  extern int                  Nukns, steady_code ;
13  extern struct elmpointers   ep ;
14  extern struct transient           tvals ;
15  extern struct eqnset        eqnsets[4] ;
16  extern double               uchange ;
17
18
19  int           varindex(ndindex,var)
20  int           ndindex, var ;
21
22  {
23        int     i, vi ;
24
25        vi = gp.iptrs[ndindex]->ui ;
26        for(i=0;i<var;i++) {
27              if(isvarfixed(ndindex,i) == 0)
28                    vi++ ;
29        }
30        return(vi) ;
31  }
32
33  int           isvarfixed(ndindex,var)
34  int           ndindex, var ;
35
36  {
37        int     fixcode, rc ;
38
39        rc = 0 ;
40        fixcode = (1 << var) ;
41        if( (fixcode & gp.iptrs[ndindex]->fxc) != 0)
42              rc = -1 ;
43        return(rc) ;
44  }
45
46  int           solve(nvar,code,nset)
47  int           code, nvar, nset ;
48
49  {
50        int     i, j, ind, solvecode ;
51        double *dp ;
52
53        if(code > 1)
54              solvecode = 1 ;
55        else
56              solvecode = code ;
57        if(code < 0) {
58              solvecode = 2 ;
59              code = - code ;
60        }
61        if(steady_code == 1)
62              printf(" Now solving ...") ;
63
```

```
64          if((N.sym ==  0)  ||  (code == 3))
65                  actcol(eqnsets[nset].Kp,eqnsets[nset].Fp,eqnsets[nset].diagp,
66      eqnsets[nset].neqns, solvecode) ;
67          else
68
69          uactcl(eqnsets[nset].Kp,eqnsets[nset].Lp,eqnsets[nset].Fp,eqnsets[nset].
70      diagp, eqnsets[nset].neqns, solvecode) ;
71
72          if(steady_code == 1)
73                  printf(" All done !\n") ;
74
75
76          dp = eqnsets[nset].Fp ;
77          if((code > 0) && (code != 3)) {
78                  if(nvar >= 0) {
79                          ind = 0 ;
80                  /*      uchange = 0.0 ;       */
81                          for(i=0;i<N.nodes;i++) {
82                                  if(code == 1) {
83                                          uchange += fabs(gp.iptrs[i]->u[nvar] - *dp)
84      ;
85                                          gp.iptrs[i]->u[nvar] = *dp ;
86                                  }
87                                  if(code == 2) {
88                                          *(eqnsets[nset+1].Fp + ind) =
89      *(eqnsets[nset].Fp + gp.iptrs[i]->i) ;
90                                  /*
91                                          gp.iptrs[i]->p[4] = *(eqnsets[nset].Fp +
92      gp.iptrs[i]->i) ;
93                                  */
94                                  }
95                                  if(code == 4) {
96                                  /*
97                                          printf(" %f  %f\n",gp.iptrs[i]->u[nvar],
98      *dp);
99                                  */
100                                         uchange += fabs(*dp) ;
101                                         gp.iptrs[i]->u[nvar] += *dp ;
102                                 }
103                                 dp++;
104                                 ind++ ;
105                         }
106                 }
107                 else {
108                         for(i=0;i<N.nodes;i++) {
109                                 for(j=0;j<N.vars;j++) {
110                                         if(code == 1)
111                                                 gp.iptrs[i]->u[j] = *dp ;
112                                         if(code == 2)
113                                                 gp.iptrs[i]->u[j] += *dp ;
114                                         dp++;
115                                 }
116                         }
117                 }
118         }
119         return(0) ;
120 }
121
122 int             actcol(Kp,Fp,diagp,neqns,code)
123 double          *Kp, *Fp ;
124 long            *diagp ;
125 int             neqns, code ;
126
```

```
127  {
128          int    i, j, nf, id, n, ni, nj ;
129          double dot(), *ip, *idp, t ;
130          char   m1[255], m2[255], m3[255], m4[255] ;
131
132          if(steady_code == 1){
133                  sprintf(m1," Now factoring equation # ") ;
134                  sprintf(m2," 0 ") ;
135                  sprintf(m3," ") ;
136                  sprintf(m4," ") ;
137                  StartMes(m1,m2,m3,m4) ;
138          }
139
140
141          for(j=1;j<neqns;j++) {
142
143                  if( (j % 10 == 0) && (steady_code == 1)) {
144                          sprintf(m4,"  %d ",j) ;
145                          UpDateMes(m1,m2,m3,m4) ;
146                  }
147
148                  nf = *(diagp+j) - *(diagp+j-1) ;
149                  if(code < 2) {
150                          ip = Kp + *(diagp+j-1) + 1 ;
151                          id = j - nf + 1 ;
152
153                          for(i=1;i<nf;i++) {
154                                  n = ((nj = i-1) < (ni = *(diagp+id) - *(diagp+id-1)
155  - 1)) ? nj : ni ;
156                                  idp = Kp + *(diagp+id++) ;
157                                  *ip++ -= dot(ip-n-1,idp-n,n) ;
158                          }
159                          ip = Kp + *(diagp+j-1) + 1 ;
160                          id = j - nf + 1 ;
161                          idp = Kp + *(diagp+j) ;
162                          for(i=1;i<nf;i++) {
163                                  t = *ip ;
164                                  *ip /= *(Kp + *(diagp+id++)) ;
165                                  *idp -= t * *ip++ ;
166                          }
167                  }
168                  ip = Kp + *(diagp+j-1) + 1 ;
169                  id = j - nf + 1 ;
170                  if(code > 0)
171                          *(Fp + j) -= dot(ip,Fp+id,nf-1) ;
172          }
173          if(code == 0) {
174                  EndMes() ;
175                  return(0) ;
176          }
177          for(j=0;j<neqns;j++)
178                  *(Fp + j) /= *(Kp + *(diagp+j)) ;
179
180          for(j=neqns-1;j>-1;j--) {
181                  if(j == 0)
182                          nf = 1;
183                  else
184                          nf = *(diagp+j) - *(diagp+j-1) ;
185                  for(i=1;i<nf;i++) {
186                          *(Fp + j - nf + i) -= *(Fp + j) * *(Kp + *(diagp+j-1) + i)
187  ;
188                  }
189          }
```

```
190
191         if(code < 2)
192                 EndMes() ;
193         return(0) ;
194 }
195
196 int             uactcl(Kp,Lp,Fp,diagp,neqns,code)
197 double          *Kp, *Lp, *Fp ;
198 long            *diagp ;
199 int             neqns, code ;
200
201 {
202         int     i, j, nf, id, n, ni, nj ;
203         double dot(), *ip, *iLp, *idp, t ;
204         char    m1[255], m2[255], m3[255], m4[255] ;
205
206         if(steady_code == 1) {
207                 sprintf(m1," Now factoring equation # ") ;
208                 sprintf(m2," 0 ") ;
209                 sprintf(m3," ") ;
210                 sprintf(m4," ") ;
211                 StartMes(m1,m2,m3,m4) ;
212         }
213
214         for(j=1;j<neqns;j++) {
215
216                 if( (j % 10 == 0) && (steady_code == 1)) {
217                         sprintf(m4,"  %d ",j) ;
218                         UpDateMes(m1,m2,m3,m4) ;
219                 }
220
221                 nf = *(diagp+j) - *(diagp+j-1) ;
222                 if(code < 2) {
223                         iLp = Lp + *(diagp+j-1) + 1 ;
224                         id = j - nf + 1 ;
225
226                         for(i=0;i<nf-1;i++) {
227                                 n = ((nj = i) < (ni = *(diagp+id) - *(diagp+id-1) -
228 1)) ? nj : ni ;
229                                 idp = Kp + *(diagp+id) ;
230                                 id++ ;
231                                 *iLp = (*iLp - dot(iLp-n,idp-n,n))/ *idp ;
232                                 iLp++ ;
233                         }
234                         ip = Kp + *(diagp+j-1) + 1 ;
235                         id = j - nf + 1 ;
236
237                         for(i=0;i<nf;i++) {
238                                 n = ((nj = i) < (ni = *(diagp+id) - *(diagp+id-1) -
239 1)) ? nj : ni ;
240                                 idp = Lp + *(diagp+id) ;
241                                 id++ ;
242                                 *ip -= dot(ip-n,idp-n,n) ;
243                                 ip++ ;
244                         }
245                 }
246                 iLp = Lp + *(diagp+j-1) + 1 ;
247                 id = j - nf + 1 ;
248                 if(code > 0)
249                         *(Fp + j) -= dot(iLp,Fp+id,nf-1) ;
250         }
251         if(code == 0) {
252                 EndMes() ;
```

```
253                 return(0) ;
254         }
255         for(j=neqns-1;j>-1;j--) {
256                 *(Fp + j) /= *(Kp + *(diagp+j)) ;
257                 if(j == 0)
258                         nf = 1;
259                 else
260                         nf = *(diagp+j) - *(diagp+j-1) ;
261                 for(i=1;i<nf;i++) {
262                         *(Fp + j - nf + i) -= *(Fp + j) * *(Kp + *(diagp+j-1) + i)
263 ;
264                 }
265         }
266         if(code < 2)
267                 EndMes() ;
268         return(0) ;
269 }
270
271
272 double dot(p1,p2,n)
273 double *p1, *p2 ;
274 int    n ;
275
276 {
277         register double      t=0.0 ;
278
279         while(n-- > 0) {
280                         t += *p1++ * *p2++ ;
281         }
282         return(t) ;
283 }
284
285 int                  get_Derivs(elmntp,pt,nvar,ddx,ddy,ddz)
286 struct element              *elmntp ;
287 struct gausspts      *pt ;
288 int                  nvar ;
289 double               *ddx, *ddy, *ddz ;
290
291 {
292         struct shapefuncs    fv;
293         struct node          *np[MNSF] ;
294         struct govfuncs             gf ;
295         int                  i, j, ns, ni, nm ;
296
297         ns =  nsf(elmntp->vtype) ;
298         for(j=0;j<elmntp->nnds;j++)
299                 np[j] = elmntp->nps[j] ;
300         get_shape(elmntp->vtype,pt,&fv) ;
301         get_gf(elmntp,&gf,&fv,&fv,np,pt->w) ;
302         *ddx = *ddy = *ddz = 0.0 ;
303         for(j=0;j<elmntp->nnds;j++) {
304                 *ddx += gf.dfdx[j] * np[j]->u[nvar] ;
305                 *ddy += gf.dfdy[j] * np[j]->u[nvar] ;
306                 *ddz += gf.dfdz[j] * np[j]->u[nvar] ;
307         }
308         *ddx *= gf.p[0] ;
309         *ddy *= gf.p[0] ;
310         *ddz *= gf.p[0] ;
311         return(ns) ;
312 }
313
314 int                  get_gf(elp,gfp,fvp,fgp,np,w)
315 struct element              *elp ;
```

```
316   struct shapefuncs     *fvp, *fgp ;
317   struct node           *np[MNSF] ;
318   struct govfuncs               *gfp ;
319   double                w ;
320
321   {
322         double J[NDIMS][NDIMS], get_J() ;
323         int    i, j ;
324
325         gfp->dof = fvp->dof ;
326
327         for(i=0;i<N.params;i++) {
328               gfp->p[i] = elp->p[i] ;
329               for(j=0;j<fvp->dof;j++)
330                     gfp->p[i] += fvp->f[j] * np[j]->p[i] ;
331         }
332         for(j=0;j<fvp->dof;j++)
333               gfp->f[j] = fvp->f[j]  ;
334
335         gfp->detJ = get_J(np,fgp,J) ;
336         gfp->w = w ;
337         invertJ(J,gfp->detJ) ;
338
339         for(j=0;j<fvp->dof;j++) {
340
341               switch (N.dims) {
342
343                     case 1 :
344                           gfp->dfdx[j] = J[0][0] * fvp->dfdr[j] ;
345                           gfp->dfdy[j] = 0.0 ;
346                           gfp->dfdz[j] = 0.0 ;
347                           break ;
348
349                     case 2 :
350                           gfp->dfdx[j] = J[0][0] * fvp->dfdr[j] + J[0][1] *
351   fvp->dfds[j] ;
352                           gfp->dfdy[j] = J[1][0] * fvp->dfdr[j] + J[1][1] *
353   fvp->dfds[j] ;
354                           gfp->dfdz[j] = 0.0 ;
355                           break ;
356
357                     case 3 :
358                           gfp->dfdx[j] = J[0][0] * fvp->dfdr[j] + J[0][1] *
359   fvp->dfds[j] + J[0][2] *fvp->dfdt[j] ;
360                           gfp->dfdy[j] = J[1][0] * fvp->dfdr[j] + J[1][1] *
361   fvp->dfds[j] + J[1][2] *fvp->dfdt[j] ;
362                           gfp->dfdz[j] = J[2][0] * fvp->dfdr[j] + J[2][1] *
363   fvp->dfds[j] + J[2][2] *fvp->dfdt[j] ;
364                           break ;
365               }
366         }
367         gfp->detJ *= gfp->w ;
368         return(0) ;
369   }
370
371   double                get_J(np,fgp,J)
372   struct shapefuncs     *fgp ;
373   struct node           *np[MNSF] ;
374   double                J[NDIMS][NDIMS] ;
375
376   {
377         int    i,j ;
378
```

```
379         for(i=0;i<N.dims;i++)
380                 for(j=0;j<N.dims;j++)
381                         J[i][j] = 0.0 ;
382
383         for(i=0;i<fgp->dof;i++)
384                 J[0][0] += fgp->dfdr[i] * np[i]->x[0] ;
385         if(N.dims == 1)
386                 return( J[0][0] ) ;
387
388         for(i=0;i<fgp->dof;i++) {
389                 J[0][1] += fgp->dfdr[i] * np[i]->x[1] ;
390                 J[1][0] += fgp->dfds[i] * np[i]->x[0] ;
391                 J[1][1] += fgp->dfds[i] * np[i]->x[1] ;
392         }
393         if(N.dims == 2)
394                 return( J[0][0] * J[1][1] - J[0][1] * J[1][0] ) ;
395
396         for(i=0;i<fgp->dof;i++) {
397                 J[0][2] += fgp->dfdr[i] * np[i]->x[2] ;
398                 J[1][2] += fgp->dfds[i] * np[i]->x[2] ;
399                 J[2][2] += fgp->dfdt[i] * np[i]->x[2] ;
400                 J[2][0] += fgp->dfdt[i] * np[i]->x[0] ;
401                 J[2][1] += fgp->dfdt[i] * np[i]->x[1] ;
402         }
403         return( J[0][0] * (J[1][1] * J[2][2] - J[1][2] * J[2][1] )
404             -   J[0][1] * (J[1][0] * J[2][2] - J[2][0] * J[1][2] )
405                 +   J[0][2] * (J[1][0] * J[2][1] - J[1][1] * J[2][1] ) ) ;
406 }
407
408 int             invertJ(J,detJ)
409 double          J[NDIMS][NDIMS], detJ ;
410
411 {
412         double K[NDIMS][NDIMS] ;
413         int    i,j ;
414
415         for(i=0;i<N.dims;i++)
416                 for(j=0;j<N.dims;j++)
417                         K[i][j] = J[i][j] ;
418
419         switch (N.dims) {
420
421                 case 1 :
422                         J[0][0] = 1.0 / K[0][0] ;
423                         break ;
424
425                 case 2 :
426                         J[0][0] =  K[1][1] / detJ ;
427                         J[0][1] = -K[0][1] / detJ ;
428                         J[1][0] = -K[1][0] / detJ ;
429                         J[1][1] =  K[0][0] / detJ ;
430                         break ;
431
432                 case 3 :
433                         J[0][0] = ( K[1][1] * K[2][2] - K[1][2] * K[2][1] ) / detJ
434 ;
435                         J[0][1] = ( K[0][2] * K[2][1] - K[0][1] * K[2][2] ) / detJ
436 ;
437                         J[0][2] = ( K[0][1] * K[1][2] - K[1][1] * K[0][2] ) / detJ
438 ;
439                         J[1][0] = ( K[1][2] * K[2][0] - K[1][0] * K[2][2] ) / detJ
440 ;
```

```
441                     J[1][1] = ( K[0][0] * K[2][2] - K[0][2] * K[2][0] ) / detJ
442     ;
443                     J[1][2] = ( K[1][0] * K[0][2] - K[0][0] * K[1][2] ) / detJ
444     ;
445                     J[2][0] = ( K[1][0] * K[2][1] - K[1][1] * K[2][0] ) / detJ
446     ;
447                     J[2][1] = ( K[0][1] * K[2][0] - K[0][0] * K[2][1] ) / detJ
448     ;
449                     J[2][2] = ( K[0][0] * K[1][1] - K[0][1] * K[1][0] ) / detJ
450     ;
451                     break ;
452             }
453             return(0) ;
454     }
455
456
457     int                     get_bKe(elmntp,theElp,nvar,code,ntf)
458     struct belement                 *elmntp, *theElp ;
459     int                     nvar, code, *ntf ;
460
461     {
462
463             struct gausspts         g[MGPTS] ;
464             struct shapefuncs       fv, ft, *fgp ;
465             struct node             *np[MNSF] ;
466             struct govfuncs                 gf, gtf ;
467             int                     i, j, ig, ns, ni, dir ;
468             float                   *fptr ;
469             double                  J00, J01, dS, tdt, mtdt ;
470
471             if(nvar < 0)
472                     ns =  nsf(elmntp->vtype) * N.vars ;
473             else
474                     ns =  nsf(elmntp->vtype) ;
475             initm(ep.K,ep.n*ep.n) ;
476             if(N.trans > 0)
477                     initm(ep.S,ep.n*ep.n) ;
478             initm(ep.F,ep.n) ;
479             *theElp = *elmntp ;
480             if(code > -1 ) {
481                     ni = get_gspts(elmntp->vtype,g) ;
482                     for(j=0;j<elmntp->nnds;j++)
483                             np[j] = elmntp->nps[j] ;
484
485                     for(ig=0;ig<ni;ig++) {
486                             get_shape(elmntp->vtype,&g[ig],&fv) ;
487                             get_shape(elmntp->gtype,&g[ig],&ft) ;
488                             if(nvar < 0)
489                                     *ntf = ft.dof * N.vars ;
490                             else
491                                     *ntf = ft.dof ;
492                             if((elmntp->bcs[nvar] == 3) && (nvar >= 0)) {
493                                     for(i=0;i<*ntf;i++) {
494                                             KE(i,i) = 1.0e16 ;
495                                             FE(i) = 1.0e16 * np[i]->u[nvar] ;
496                                     }
497                                     break ;
498                             }
499
500                             J00 = J01 = 0.0 ;
501                             fgp = &fv ;
502                             for(i=0;i<fgp->dof;i++) {
503                                     J00 += fgp->dfdr[i] * np[i]->x[0] ;
```

```
504                             J01 += fgp->dfdr[i] * np[i]->x[1] ;
505                     }
506                     dS = g[ig].w * sqrt(J00 * J00 + J01 * J01) ;
507                     update_BKe(elmntp,&fv,&ft,ns,*ntf,np,nvar,dS) ;
508             }
509     }
510     if(N.trans > 0) {
511             tdt = tvals.theta * tvals.dt ;
512             mtdt = tvals.dt * (1.0 - tvals.theta) ;
513             for(i=0;i<*ntf;i++) {
514                     FE(i) *= tvals.dt ;
515                     for(j=0;j<ns;j++) {
516                             FE(i) += (-mtdt * KE(i,j)) * (elmntp->nps[j])-
517 >u[nvar] ;
518                             SE(i,j) = tdt * KE(i,j) ;
519                     }
520             }
521     }
522     return(ns) ;
523 }
524
525
526 int     initm(K,n)
527 double *K ;
528 int     n ;
529
530 {
531
532     register int  i ;
533
534     for(i=0;i<n;i++)
535             *(K++) = 0.0 ;
536     return(0) ;
537 }
538
539
540 int         NdinEl(n,elpp)
541 int         n ;
542 struct element      *(*elpp) ;
543
544 {
545     int    i, rc;
546     struct element      *selp ;
547
548     rc = 0 ;
549     selp = *elpp ;
550     while(selp != NULL) {
551             for(i=0;i<selp->nnds;i++) {
552                     if(n == (selp->nps[i])->n) {
553                             rc = 1 ;
554                             *elpp = selp ;
555                             break ;
556                     }
557             }
558             if(rc == 1)
559                     return(i) ;
560             selp = selp->nextelp ;
561     }
562     return(-1) ;
563 }
564
565
566 int         get_cline(elp,cval,niv,nvar,tol,xl,yl)
```

```
int           niv, nvar ;
double        xl[], yl[], tol, cval ;
struct element       *elp ;

{
      int                  i, j, nipts, rc ;
      double               x, y, Dx, Dy, dfdr, dfds ;
      struct node          *np[MNSF] ;
      struct shapefuncs    sf ;
      struct gausspts            xg ;

      if(niv == 0)
            tol = -1.0 ;
      nipts = isvalinel(elp,np,cval,nvar,tol,&dfdr,&dfds,xl,yl) ;
      switch (nipts) {
            case  0 :
                  break ;
            case 2 :
                  Dx = xl[1] - xl[0] ;
                  Dy =  yl[1] - yl[0] ;
                  xl[niv+1] = xl[1] ;
                  yl[niv+1] = yl[1] ;
                  for(i=1;i<=niv;i++) {
                        xl[i] = xl[i-1] + (xl[niv+1] - xl[i-1]) * 1 / (niv
- i + 2) ;
                        yl[i] = yl[i-1] + (yl[niv+1] - yl[i-1]) * 1 / (niv
- i + 2) ;
                        find_2Dloc(np,nvar+N.dims,elp-
>gtype,cval,tol,Dx,Dy,dfdr,dfds,&xl[i],&yl[i]) ;
                  }
                  xg.z = 0.0 ;
                  xg.w = -1.0 ;
                  for(i=0;i<=niv+1;i++) {
                        xg.x = xl[i] ;
                        xg.y = yl[i] ;
                        rc = get_shape(elp->gtype,&xg,&sf) ;
                        xl[i] = yl[i] = 0.0 ;
                        for(j=0;j<sf.dof;j++) {
                              xl[i] += sf.f[j] * np[j]->x[0] ;
                              yl[i] += sf.f[j] * np[j]->x[1] ;
                        }
                  }
                  break ;
      }

      return(nipts) ;
}

int           isvalinel(elp,np,cval,nvar,tol,dfdr,dfds,xl,yl)
int           nvar ;
double        xl[], yl[], tol, cval, *dfdr, *dfds ;
struct node   *np[] ;
struct element       *elp ;

{
      int           j, k, nbps, etyp, ng, next, prev, ncpts, aj[MNSF],
ak[MNSF] ;
      double        Dx, Dy, r, s, t, f, fr, fs, ft ;
      struct gausspts      g[MGPTS] ;

      nbps = nbounds(elp->gtype) ;
      etyp = elp->gtype ;
      for(j=0;j<elp->nnds;j++)
```

```
630                 np[j] =  elp->nps[j] ;
631           prev = (cval <= np[nbps-1]->u[nvar]) ? 1 : -1 ;
632           ncpts = 0 ;
633           for(j=0;j<nbps;j++) {
634                 next = (cval <= np[j]->u[nvar]) ? 1 : -1 ;
635                 if(next != prev) {
636                       aj[ncpts] = j ;
637                       if( j == 0 )
638                             ak[ncpts] = nbps-1 ;
639                       else
640                             ak[ncpts] = j - 1 ;
641                       ncpts++ ;
642                 }
643                 prev = next ;
644           }
645           if((ncpts > 0) && (tol > 0.0) ) {
646                 ng = get_gspts(etyp,g) ;
647                 *dfdr = *dfds = 0.0 ;
648                 for(j=0;j<ng;j++) {
649                       r = g[j].x ;
650                       s = g[j].y ;
651                       t = g[j].z ;
652                       get_value(np,nvar,etyp,r,s,t,2,&f,&fr,&fs,&ft) ;
653                       *dfdr += fr ;
654                       *dfds += fs ;
655                 }
656                 *dfdr /= ng ;
657                 *dfds /= ng ;
658           }
659           for(j=0;j<ncpts;j++) {
660
661           GetIPoint(aj[j],ak[j],np,etyp,cval,nvar,&xl[j],&yl[j],&Dx,&Dy) ;
662                       if(tol > 0.0)
663
664           find_2Dloc(np,nvar+N.dims,etyp,cval,tol,Dx,Dy,*dfdr,*dfds,&xl[j],&yl[j])
665     ;
666           }
667           return(ncpts) ;
668     }
669
670     int           GetIPoint(j,k,np,eltype,u,n,xp,yp,Dx,Dy)
671     double        *xp, *yp, *Dx, *Dy, u ;
672     int           j, k, n, eltype;
673     struct node   *np[] ;
674
675     {
676           double x1, y1, x2, y2 ;
677
678           get_nlcs(eltype,k,&x1,&y1) ;
679           get_nlcs(eltype,j,&x2,&y2) ;
680           *xp = x1 + (u - np[k]->u[n])/(np[j]->u[n] - np[k]->u[n])*(x2 - x1) ;
681           *yp = y1 + (u - np[k]->u[n])/(np[j]->u[n] - np[k]->u[n])*(y2 - y1) ;
682           *Dx = y2 - y1 ;
683           *Dy = x1 - x2 ;
684           return(0) ;
685     }
686
687     int           find_2Dloc(np,nvar,eltype,fc,tol,Dr,Ds,dfdr,dfds,r,s)
688     double        fc, tol, Dr, Ds, dfdr, dfds, *r, *s ;
689     int           eltype, nvar ;
690     struct node   *np[] ;
691
692     {
```

```
693         int     i, eq, imax = 100 ;
694         double t, f, ddr, dds, dfdt, df, dr, ds, sl ;
695
696         printf(" in find_2Dloc...") ;
697
698         t = 0.0 ;
699         if(fabs(Dr) >= fabs(Ds)) {
700               eq = 2 ;
701               sl = -Ds/Dr ;
702         }
703         else {
704               eq = 1 ;
705               sl = -Dr/Ds ;
706         }
707         for(i=0;i<imax;i++) {
708               if( get_value(np,nvar,eltype,*r,*s,t,2,&f,&ddr,&dds,&dfdt)  != 0)
709                     return(-1);
710               if(fabs(df = f - fc) < tol) {
711                     printf(" fc %f f %f df %f tol %f\n",fc,f,df,tol);
712                     break ;
713               }
714               if(eq == 2) {
715                     ds = df / (ddr*sl + dds) ;
716                     dr = sl*ds ;
717               }
718               else {
719                     dr = df / (ddr + sl*dds) ;
720                     ds = sl * dr ;
721               }
722               if(fabs(dr) > 0.5 || fabs(ds) > 0.5) {
723                     if(eq == 2) {
724                           ds = df / (dfdr*sl + dfds) ;
725                           dr = sl*ds ;
726                     }
727                     else {
728                           dr = df / (dfdr + sl*dfds) ;
729                           ds = sl * dr ;
730                     }
731               }
732               if(fabs(dr) > 0.5 || fabs(ds) > 0.5)
733                     return(-2) ;
734               *r -= dr ;
735               *s -= ds ;
736               printf(" %d ",i) ;
737         }
738         printf(" ...out\n") ;
739         return(i) ;
740   }
741
742   int           get_value(np,nvar,eltype,r,s,t,code,f,dfdr,dfds,dfdt)
743   double        *f, *dfdr, *dfds, *dfdt, r, s, t ;
744   int           eltype, nvar, code ;
745   struct node   *np[] ;
746
747   {
748         int                   i, rc ;
749         struct shapefuncs     sf ;
750         struct gausspts             x ;
751
752         x.x = r ;
753         x.y = s ;
754         x.z = t ;
755         x.w = 0.0  ;
```

```
756         if(code == 0)
757               x.w = -1.0 ;
758         rc = get_shape(eltype,&x,&sf) ;
759
760         switch (code) {
761               case  3 :
762                     *dfdt = 0.0 ;
763                     for(i=0;i<sf.dof;i++)
764                           *dfdt += sf.dfdt[i] * np[i]->x[nvar] ;
765               case 2 :
766                     *dfds = 0.0 ;
767                     for(i=0;i<sf.dof;i++)
768                           *dfds += sf.dfds[i] * np[i]->x[nvar] ;
769               case 1 :
770                     *dfdr = 0.0 ;
771                     for(i=0;i<sf.dof;i++)
772                           *dfdr += sf.dfdr[i] * np[i]->x[nvar] ;
773               case 0 :
774                     *f = 0.0 ;
775                     for(i=0;i<sf.dof;i++)
776                           *f += sf.f[i] * np[i]->x[nvar] ;
777         }
778         return(rc) ;
779   }
```

```
1   #include "fe.h"
2
3   #define KE(A,B)       ( *(ep.K + ep.n*A + B) )
4   #define SE(A,B)       ( *(ep.S + ep.n*A + B) )
5   #define ME(A,B)       ( *(ep.M + ep.n*A + B) )
6   #define FE(B)               ( *(ep.F + B) )
7   #define TRUE          -1
8
9   extern struct control             N ;
10  extern struct pointers            gp;
11  extern struct RegMesh             Mesh ;
12  extern struct fxnodes             *first_fnp ;
13  extern int                        Nukns ;
14  extern struct elmpointers   ep ;
15  extern struct transient           tvals ;
16  extern struct RegMesh             Mesh ;
17  extern unsigned long              d_time ;
18
19
20  int           list_vars(nvar)
21  int           nvar ;
22
23  {
24          int           i, j ;
25          struct node   *np ;
26          FILE   *fp, *put_fptr() ;
27
28          fp = put_fptr(1)  ;
29
30          fprintf(fp," Node\t Value \n\n");
31
32          for(i=0;i<N.nodes;i++) {
33
34                        for(j=0;j<N.dims;j++)
35                              put_dbl(fp,(double) gp.iptrs[i]->x[j]) ;
36                        for(j=0;j<N.vars;j++)
37                              put_dbl(fp,(double) gp.iptrs[i]->u[j]) ;
38                        fprintf(fp,"\n") ;
39
40          }
41          printf("ok after variables\n") ;
42          fclose(fp) ;
43          return ;
44  }
45
46  int           test_contour(code)
47  int           code ;
48
49  {
50          int     nelm,niv,nvar, ni, i ;
51          double tol, xl[10], yl[10], cval ;
52          struct element        *elp ;
53
54          printf(" Input el#, #iv, #var, value, tol \n") ;
55          scanf(" %d %d %d %lf %lf",&nelm,&niv,&nvar,&cval,&tol) ;
56          elp = gp.El ;
57          for(i=0;i<N.elms;i++) {
58                  if(elp->n == nelm)
59                          break ;
60                  elp = elp->nextelp ;
61          }
62          ni = get_cline(elp,cval,niv,nvar,tol,xl,yl) ;
63          printf(" number of intersections = %d\n",ni) ;
```

```
64          if(ni == 2) {
65                  printf(" contour line\n") ;
66                  for(i=0;i<niv+2;i++)
67                          printf(" %f  %f \n",xl[i],yl[i]) ;
68          }
69          return(0) ;
70  }
71
72
73  int             test_trans(nvar)
74  int             nvar ;
75
76  {
77          int     i, nt ;
78
79          printf(" Time = %lf \n", tvals.t) ;
80          printf(" Input theta, delta t and magnification factor \n") ;
81          scanf(" %lf %lf %lf",&tvals.theta,&tvals.dt,&tvals.dtfac) ;
82          printf(" Input number of time steps \n") ;
83          scanf(" %d",&tvals.nsteps) ;
84          transient(0,1) ;
85          return(0) ;
86  }
87
88  int             test_mesh(eltype)
89  int             eltype ;
90
91  {
92          int                             i, ni ;
93          struct element                  *elp ;
94
95          Mesh.eltype = eltype ;
96          printf(" Input nx, nbx, ny, nby \n") ;
97          scanf(" %d %d %d %d",&Mesh.nx,&Mesh.nbx,&Mesh.ny,&Mesh.nby) ;
98          ni = MkLapMesh(&Mesh) ;
99          printf(" return code = %d\n",ni) ;
100         return(0) ;
101 }
102
103
104 int             test_map(maptype)
105 int             maptype ;
106
107 {
108         int                             i, n, k, l, numrows, numcols ;
109         struct blockmap         *theblock ;
110
111         Mesh.maptype = maptype ;
112         theblock = Mesh.firstmap ;
113         for(k=0;k<Mesh.nby;k++) {
114                 for(l=0;l<Mesh.nbx;l++) {
115                         printf(" Input macro element nodes x,y coord. pairs\n") ;
116                         n = nsf(maptype) ;
117                         for(i=0;i<n;i++) {
118                                 scanf(" %f %f",&(theblock->xnodes[i]),&(theblock-
119 >ynodes[i])) ;
120                         }
121                         map_mesh(theblock,Mesh.maptype,k,l) ;
122                         theblock++ ;
123                 }
124         }
125         return(0) ;
126 }
```

```
127
128   int           set_bc(code)
129   int           code ;
130
131   {
132         int                         bestart, beend,i, j, bctype ;
133         double                bvalue ;
134         struct belement       *belp ;
135
136         while (TRUE) {
137               printf(" Input first, last boundary elements, code and value\n")
138   ;
139               scanf(" %d %d %d %lf",&bestart,&beend,&bctype,&bvalue) ;
140               if(bestart <= 0)
141                     break ;
142               belp = gp.B ;
143               for(i=0;i<N.belms;i++) {
144                     if(belp->n == bestart)
145                           break ;
146                     belp = belp->nextbelp ;
147               }
148               do {
149                     if(bctype == 0) {
150                           belp->bcs[0] = 0 ;
151                           belp->bcs[1] = 1 ;
152                           for(j=0;j<belp->nnds;j++) {
153                                 (belp->nps[j])->u[1] = bvalue ;
154                           }
155                     }
156                     else {
157                           belp->bcs[0] = 1 ;
158                           belp->bcs[1] = 0 ;
159                           for(j=0;j<belp->nnds;j++) {
160                                 (belp->nps[j])->u[0] = bvalue ;
161                           }
162                     }
163                     if(belp->n == beend)
164                           break;
165                     belp = belp->nextbelp ;
166               }
167               while(belp->n <= N.belms) ;
168         }
169         return(0) ;
170   }
171
172
173   int    test_real_stream(eltype)
174   int eltype ;
175   {
176         int                                 i, j, k, l, numrows, numcols, el,
177   n_x, n_z;
178         int                                 maptype, get_int(), is, iend, off_set
179   ;
180         int                                 actual_nbx ;
181         struct node                   *np, *node_u ;
182         struct blockmap               *theblock ;
183         FILE                          *f, *get_fptr(),*put_fptr() ;
184         struct cross_section          *x_all,  *x_c, *x_n, *x_n_n ;
185         double                              get_dbl() ;
186         double                              diag_l, dx_ave, dy_ave, co, si,
187   E_long, E_lat ;
188         double                              x1 , x0, y1, y0, x2, x3, y2, y3;
189         double                              constant = 100.0 ;
```

```
190         char                        smooth_code = 'n' ;
191
192         printf( " Input the type of mapping desired (211, 221)\n->");
193         scanf(" %d",&maptype) ;
194         if(maptype == 211){
195               off_set  = 0 ;
196               printf(" Using linear mapping\n");
197         }
198         else {
199               printf(" Using quadratic mapping\n");
200               off_set = 1 ;
201         }
202         printf(" Do   you want smoothing ? \n  ->") ;
203         scanf(" %c",&smooth_code) ;
204
205         f = get_fptr(1) ;
206
207         Mesh.eltype = eltype ;
208         Mesh.maptype = maptype ;
209         Mesh.nx = get_int(f) ;
210         actual_nbx = get_int(f) ;
211         if((x_all = (struct cross_section *) lmalloc(
212         (long int)(actual_nbx+1)*sizeof(struct cross_section) )) == NULL) /*
213         NON STANDARD FUNCTION       */
214               return(-1)  ;
215         if( actual_nbx % 2 == 0 && maptype == 221)
216               Mesh.nbx = actual_nbx / 2 ;
217         else
218               Mesh.nbx = actual_nbx ;
219         Mesh.ny = get_int(f) ;
220         Mesh.nby = get_int(f) ;
221         free(Mesh.firstmap);
222         Mesh.firstmap = (struct blockmap *)calloc(Mesh.nbx*Mesh.nby
223                                                         ,sizeof(struct
224   blockmap)) ;
225
226         printf(" Generation a mesh with %d by %d blocks\n",Mesh.nbx,Mesh.nby) ;
227         ni = MkLapMesh(&Mesh) ;
228
229         if(ni != 0)
230               exit(0) ;
231
232         if(maptype == 221 && (actual_nbx % 2 != 0))
233               ni =First_map(&Mesh) ;
234
235         if(maptype == 221){
236               if(actual_nbx % 2 == 0)
237                     off_set = 0 ;
238               else {
239                     Mesh.nbx /= 2 ;
240                     off_set = 1 ;
241                     Mesh.nbx += off_set ;
242               }
243         }
244
245         n_x = actual_nbx + 1 ;
246         n_z = Mesh.nby + 1 ;
247         for(i=0;i<n_x;i++){
248               x_c = x_all + i ;
249               for(j=0;j<n_z;j++){
250                     x_c->x[j] = get_dbl(f)+constant ;
251                     x_c->z[j] = get_dbl(f)+constant ;
252                     x_c->h[j] = get_dbl(f)+constant ;
```

```
253                     x_c->u[j] = get_dbl(f)+constant ;
254                     x_c->v[j] = get_dbl(f)+constant ;
255                     x_c->E_long[j] = get_dbl(f)+constant ;
256                     x_c->E_lat[j] = get_dbl(f)+constant;
257             }
258             x_c->next_x = x_c + 1;
259     } .
260
261     fclose(f) ;
262     Mesh.maptype = maptype ;
263     theblock = Mesh.firstmap ;
264     x_c = x_all ;
265
266
267     for(k=0;k<Mesh.nbx;k++) {
268             x_n = x_c->next_x ;
269
270             if(k > (Mesh.nbx-2) && off_set == 1)
271                     Mesh.maptype = 211 ;
272
273             printf(" mapping type = %d\n",Mesh.maptype) ;
274
275             for(l=0;l<Mesh.nby;l++) {
276                     switch(Mesh.maptype){
277                             case  211 :
278                                     theblock->xnodes[0] = x_n->x[l] ;
279                                     theblock->xnodes[1] = x_n->x[l+1] ;
280                                     theblock->xnodes[2] = x_c->x[l+1] ;
281                                     theblock->xnodes[3] = x_c->x[l] ;
282
283                                     theblock->ynodes[0] = x_n->z[l] ;
284                                     theblock->ynodes[1] = x_n->z[l+1] ;
285                                     theblock->ynodes[2] = x_c->z[l+1] ;
286                                     theblock->ynodes[3] = x_c->z[l] ;
287
288                                     theblock->h[0] = x_n->h[l] ;
289                                     theblock->h[1] = x_n->h[l+1] ;
290                                     theblock->h[2] = x_c->h[l+1] ;
291                                     theblock->h[3] = x_c->h[l] ;
292
293                                     theblock->u[0] = x_n->u[l] ;
294                                     theblock->u[1] = x_n->u[l+1] ;
295                                     theblock->u[2] = x_c->u[l+1] ;
296                                     theblock->u[3] = x_c->u[l] ;
297
298                                     theblock->v[0] = x_n->v[l] ;
299                                     theblock->v[1] = x_n->v[l+1] ;
300                                     theblock->v[2] = x_c->v[l+1] ;
301                                     theblock->v[3] = x_c->v[l] ;
302
303                                     theblock->E_x[0] = x_n->E_long[l] ;
304                                     theblock->E_x[1] = x_n->E_long[l+1] ;
305                                     theblock->E_x[2] = x_c->E_long[l+1] ;
306                                     theblock->E_x[3] = x_c->E_long[l] ;
307
308                                     theblock->E_y[0] = x_n->E_lat[l] ;
309                                     theblock->E_y[1] = x_n->E_lat[l+1] ;
310                                     theblock->E_y[2] = x_c->E_lat[l+1] ;
311                                     theblock->E_y[3] = x_c->E_lat[l] ;
312
313                                     theblock->E_xy[0] = constant ;
314                                     theblock->E_xy[1] = constant ;
315                                     theblock->E_xy[2] = constant ;
```

```
316                                     theblock->E_xy[3] = constant ;
317
318                                     theblock->E_yx[0] = constant ;
319                                     theblock->E_yx[1] = constant ;
320                                     theblock->E_yx[2] = constant ;
321                                     theblock->E_yx[3] = constant ;
322                                     break ;
323
324                                 case 221 :
325                                     x_n_n = x_n->next_x ;
326
327                                     theblock->xnodes[0] = x_n_n->x[1] ;
328                                     theblock->xnodes[1] = (x_n_n->x[1]+x_n_n-
329     >x[1+1])/2. ;
330                                     theblock->xnodes[2] = x_n_n->x[1+1] ;
331                                     theblock->xnodes[3] = x_n->x[1+1] ;
332                                     theblock->xnodes[4] = x_c->x[1+1] ;
333                                     theblock->xnodes[5] = (x_c->x[1+1] + x_c-
334     >x[1])/2. ;
335                                     theblock->xnodes[6] = x_c->x[1] ;
336                                     theblock->xnodes[7] = x_n->x[1] ;
337
338                                     theblock->ynodes[0] = x_n_n->z[1] ;
339                                     theblock->ynodes[1] = (x_n_n->z[1]+x_n_n-
340     >z[1+1])/2. ;
341                                     theblock->ynodes[2] = x_n_n->z[1+1] ;
342                                     theblock->ynodes[3] = x_n->z[1+1] ;
343                                     theblock->ynodes[4] = x_c->z[1+1] ;
344                                     theblock->ynodes[5] = (x_c->z[1+1] + x_c-
345     >z[1])/2. ;
346                                     theblock->ynodes[6] = x_c->z[1] ;
347                                     theblock->ynodes[7] = x_n->z[1] ;
348
349                                     theblock->h[0] = x_n_n->h[1] ;
350                                     theblock->h[1] = (x_n_n->h[1]+x_n_n-
351     >h[1+1])/2. ;
352                                     theblock->h[2] = x_n_n->h[1+1] ;
353                                     theblock->h[3] = x_n->h[1+1] ;
354                                     theblock->h[4] = x_c->h[1+1] ;
355                                     theblock->h[5] = (x_c->h[1+1] + x_c-
356     >h[1])/2. ;
357                                     theblock->h[6] = x_c->h[1] ;
358                                     theblock->h[7] = x_n->h[1] ;
359
360                                     theblock->u[0] = x_n_n->u[1] ;
361                                     theblock->u[1] = (x_n_n->u[1]+x_n_n-
362     >u[1+1])/2. ;
363                                     theblock->u[2] = x_n_n->u[1+1] ;
364                                     theblock->u[3] = x_n->u[1+1] ;
365                                     theblock->u[4] = x_c->u[1+1] ;
366                                     theblock->u[5] = (x_c->u[1+1] + x_c-
367     >u[1])/2. ;
368                                     theblock->u[6] = x_c->u[1] ;
369                                     theblock->u[7] = x_n->u[1] ;
370
371                                     theblock->v[0] = x_n_n->v[1] ;
372                                     theblock->v[1] = (x_n_n->v[1]+x_n_n-
373     >v[1+1])/2. ;
374                                     theblock->v[2] = x_n_n->v[1+1] ;
375                                     theblock->v[3] = x_n->v[1+1] ;
376                                     theblock->v[4] = x_c->v[1+1] ;
377                                     theblock->v[5] = (x_c->v[1+1] + x_c-
378     >v[1])/2. ;
```

```
379                                     theblock->v[6] = x_c->v[1] ;
380                                     theblock->v[7] = x_n->v[1] ;
381
382                                     theblock->E_x[0] = x_n_n->E_long[1] ;
383                                     theblock->E_x[1] = (x_n_n->E_long[1]+x_n_n-
384     >E_long[1+1])/2. ;
385                                     theblock->E_x[2] = x_n_n->E_long[1+1] ;
386                                     theblock->E_x[3] = x_n->E_long[1+1] ;
387                                     theblock->E_x[4] = x_c->E_long[1+1] ;
388                                     theblock->E_x[5] = (x_c->E_long[1+1] + x_c-
389     >E_long[1])/2. ;
390                                     theblock->E_x[6] = x_c->E_long[1] ;
391                                     theblock->E_x[7] = x_n->E_long[1] ;
392
393                                     theblock->E_y[0] = x_n_n->E_lat[1] ;
394                                     theblock->E_y[1] = (x_n_n->E_lat[1]+x_n_n-
395     >E_lat[1+1])/2. ;
396                                     theblock->E_y[2] = x_n_n->E_lat[1+1] ;
397                                     theblock->E_y[3] = x_n->E_lat[1+1] ;
398                                     theblock->E_y[4] = x_c->E_lat[1+1] ;
399                                     theblock->E_y[5] = (x_c->E_lat[1+1] + x_c-
400     >E_lat[1])/2. ;
401                                     theblock->E_y[6] = x_c->E_lat[1] ;
402                                     theblock->E_y[7] = x_n->E_lat[1] ;
403
404                                     theblock->E_xy[0] = constant ;
405                                     theblock->E_xy[1] = constant ;
406                                     theblock->E_xy[2] = constant ;
407                                     theblock->E_xy[3] = constant ;
408                                     theblock->E_xy[4] = constant ;
409                                     theblock->E_xy[5] = constant ;
410                                     theblock->E_xy[6] = constant ;
411                                     theblock->E_xy[7] = constant ;
412
413                                     theblock->E_yx[0] = constant ;
414                                     theblock->E_yx[1] = constant ;
415                                     theblock->E_yx[2] = constant ;
416                                     theblock->E_yx[3] = constant ;
417                                     theblock->E_yx[4] = constant ;
418                                     theblock->E_yx[5] = constant ;
419                                     theblock->E_yx[6] = constant ;
420                                     theblock->E_yx[7] = constant ;
421
422                                     break ;
423                             }
424
425                         map_stream(theblock,Mesh.maptype,1,k) ;
426                         theblock++ ;
427                 }
428
429         if(maptype == 221)
430                 x_c = x_n->next_x ;
431         else
432                 x_c = x_c->next_x ;
433         }
434
435         np = gp.N ;
436         for(i=0;i<N.nodes;i++){
437                 np->x[0] -= constant ;
438                 np->x[1] -= constant ;
439                 np->p[0] -= constant ;
440                 np->p[1] -= constant ;
441                 np->p[2] -= constant ;
```

```
442                 np->p[3] -= constant ;
443                 np->p[4] -= constant ;
444                 np->p[5] -= constant ;
445                 np->p[7] -= constant ;
446                 np = np->nextnp ;
447         }
448
449         ni = Get_vel_vec() ;
450
451         if( smooth_code == 'y'){
452                 printf(" Performing Smoothing !\n") ;
453                 for(i=0;i<5;i++)
454                         Smooth_Mesh(actual_nbx,Mesh.ny) ;
455         }
456
457         free(Mesh.firstmap);
458
459         return(0) ;
460 }
461
462 int             Smooth_Mesh(nx,ny)
463 int             nx, ny ;
464 {
465         register int  i,j,k ;
466         int                     index, i_1,i_2,i_3,i_4 ;
467
468         for(i=2;i< nx;i++){
469                 for(j=2;j< ny;j++){
470                         index = (i-1) * (ny+1) + j - 1 ;
471                         i_1 = i*(ny+1) + j - 1 ;
472                         i_2 = (i-1)  * (ny+1) + j ;
473                         i_3 = (i-2)*(ny+1) + j - 1 ;
474                         i_4 = (i-1) * (ny+1) + j - 2 ;
475
476                         gp.iptrs[index]->x[0] = (   gp.iptrs[i_1]->x[0] / 2.
477                                                 +       gp.iptrs[i_2]->x[0]
478                                                 +       gp.iptrs[i_3]->x[0] / 2.
479                                                 +       gp.iptrs[i_4]->x[0] ) / 3.0 ;
480                         gp.iptrs[index]->x[1] = (   gp.iptrs[i_1]->x[1]
481                                                 +       gp.iptrs[i_2]->x[1] / 2.
482                                                 +       gp.iptrs[i_3]->x[1]
483                                                 +       gp.iptrs[i_4]->x[1] / 2. ) /
484 3.0 ;
485
486                         for(k=0;k<N.params;k++){
487                                 gp.iptrs[index]->p[k] = (   gp.iptrs[i_1]->p[k]
488                                                         +       gp.iptrs[i_2]->p[k]
489                                                         +       gp.iptrs[i_3]->p[k]
490                                                         +       gp.iptrs[i_4]->p[k] ) /
491 4.0 ;
492                         }
493                 }
494         }
495         return(0) ;
496 }
497
498
499 int     Get_vel_vec()
500 {
501         int                                     i, j, k, l, ni, n_x, n_z, nm ;
502         int                                     get_int(), is, iend, off_set,
503 index,n1,n2 ;
504         register struct node            *np1, *np2 ;
```

```
505         double                              diag_1, dx_ave, dy_ave, co, si,
506   E_long, E_lat ;
507         double                              x1 , x0, y1, y0, x2, x3, y2, y3,
508   q_old;
509         register struct element     *elp ;
510         double                      tol, xl[10], yl[10], cval ;
511
512
513         for(i=1;i<=(Mesh.nx+1);i++){
514               q_old = 0.0 ;
515               for(j=1;j<=(Mesh.ny);j++){
516                     index = (i - 1) * (Mesh.ny+1) + j - 1;
517                     np1 = gp.N ;
518                     for(k=0;k<N.nodes;k++){
519                           if(np1->n == (index+1))
520                                 break ;
521                           else
522                                 np1 = np1->nextnp ;
523                     }
524                     np2 = np1->nextnp ;
525
526                     x1 = np1->x[0];
527                     x2 = np2->x[0];
528                     y1 = np1->x[1];
529                     y2 = np2->x[1];
530                     diag_1 = sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2)) ;
531                     q_old += (np1->p[0]+np2->p[0])
532                                 *diag_1*(np1->p[7]+np2->p[7])/4.0 ;
533
534                     np2->u[0] = q_old ;
535               }
536         }
537         for(i=1;i<=(Mesh.nx+1);i++){
538               q_old = 0.0 ;
539               iend = (i-1) *( Mesh.ny+1) + Mesh.ny ;
540               np2 = gp.N ;
541               for(k=0;k<N.nodes;k++){
542                     if(np2->n==(iend+1))
543                           break ;
544                     else
545                           np2 = np2->nextnp ;
546               }
547               for(j=1;j<=(Mesh.ny+1);j++){
548                     index = (i - 1) *(Mesh.ny+1) + j - 1;
549                     np1 = gp.N ;
550                     for(k=0;k<N.nodes;k++){
551                           if(np1->n==(index+1))
552                                 break;
553                           else
554                                 np1 = np1->nextnp ;
555                     }
556                     np1->u[0] /= np2->u[0] ;
557
558               }
559         }
560         printf(" Finished Obtaining q\n") ;
561
562         np1 = gp.N ;
563         for(k=0;k<N.nodes;k++){
564               np1->ui = 0 ;
565               elp = gp.El ;
566
567               for(i=0;i<N.elms;i++){
```

```
568                     for(j=0;j<elp->nnds;j++){
569                             if((elp->nps[j])->n == np1->n)
570                                     np1->ui += 1 ;
571                     }
572                     elp = elp->nextelp ;
573             }
574             np1 = np1->nextnp ;
575     }
576
577
578     tol = 0.0001 ;
579     elp = gp.El ;
580     for(i=0;i<N.elms;i++){
581             for(j=0;j<4;j++){
582                     ni = get_cline(elp,(elp->nps[j])->u[0],0,0,tol,xl,yl) ;
583                     dx_ave = xl[1] - xl[0] ;
584                     dy_ave = yl[1] - yl[0] ;
585                     if((fabs(dx_ave) > tol && fabs(dy_ave) > tol)){
586                             switch(ni){
587                                     case 2:
588                                             (elp->nps[j])->p[6] +=
589 atan(dy_ave/dx_ave) ;
590                                     break ;
591                             }
592                     }
593             }
594             elp = elp->nextelp ;
595     }
596     is = Mesh.ny + 1  ;
597     iend = N.nodes - Mesh.ny -1 ;
598
599     np2 = gp.N ;
600     for(i=0;i<iend;i++){
601             if(i > is && np2->ui != 1)
602                     np2->p[6] /= 2.   ;
603             np2 = np2->nextnp ;
604     }
605     printf(" Finished Obtaining the angles\n") ;
606
607
608
609     np2 = gp.N ;
610     for(i=0;i<N.nodes;i++){
611             co = cos(np2->p[6]) ;
612             si = sin(np2->p[6]) ;
613
614             E_long = np2->p[0] ;
615             E_lat = np2->p[1] ;
616
617             np2->p[0] = E_long * co - E_lat  * si ;
618             np2->p[1] = E_long * si + E_lat  * co ;
619
620             E_long = np2->p[2] ;
621             E_lat = np2->p[5] ;
622
623             np2->p[2] = E_long *co*co + E_lat *si*si ;
624             np2->p[3] = np2->p[4] = (E_long - E_lat) *si*co ;
625             np2->p[5] = E_long *si*si + E_lat *co*co ;
626
627             np2->u[0] = 0.0 ;
628             np2->p[6] = 0.0 ;
629
630             np2 = np2->nextnp ;
```

```
631         }
632         return(0) ;
633   }
634
635   int                           map_stream(theblock,maptype,rownum,colnum)
636   int                           maptype, rownum, colnum ;
637   struct blockmap       *theblock ;
638
639   {
640         struct node             *np ;
641         struct gausspts         g ;
642         struct shapefuncs       sf ;
643         int                             i, j ;
644
645         np = gp.N ;
646         g.z = 0.0 ;
647         g.w = -1.0 ;
648         for(i=0;i<N.nodes;i++) {
649               g.x = np->x[0] - 2.0 * colnum ;
650               g.y = np->x[1] - 2.0 * rownum ;
651               if(get_shape(maptype,&g,&sf) == 0) {
652                     np->x[0] = 0.0 ;
653                     np->x[1] = 0.0 ;
654                     np->p[0] = 0.0 ;
655                     np->p[1] = 0.0 ;
656                     np->p[2] = 0.0 ;
657                     np->p[3] = 0.0 ;
658                     np->p[4] = 0.0 ;
659                     np->p[5] = 0.0 ;
660                     np->p[7] = 0.0 ;
661
662                     for(j=0;j<sf.dof;j++) {
663                           np->x[0] += sf.f[j] * theblock->xnodes[j] ;
664                           np->x[1] += sf.f[j] * theblock->ynodes[j] ;
665                           np->p[0] += sf.f[j] * theblock->u[j] ;
666                           np->p[1] += sf.f[j] * theblock->v[j] ;
667                           np->p[2] += sf.f[j] * theblock->E_x[j] ;
668                           np->p[3] += sf.f[j] * theblock->E_xy[j] ;
669                           np->p[4] += sf.f[j] * theblock->E_yx[j] ;
670                           np->p[5] += sf.f[j] * theblock->E_y[j] ;
671                           np->p[7] += sf.f[j] * theblock->h[j] ;
672                     }
673               }
674               np = np->nextnp ;
675         }
676         return(0) ;
677   }
678
679   int           First_map(theMesh)
680   struct RegMesh *theMesh ;
681   {
682         int                             i, n, k, l, numrows, numcols, new_nbx,
683   off_set ;
684         struct blockmap       *theblock ;
685         float                 ks, ls ;
686         struct                        node    *nodep ;
687
688         theblock = (struct blockmap *)calloc((theMesh->nbx*theMesh-
689   >nby),sizeof(struct blockmap)) ;
690
691         new_nbx = theMesh->nbx/2 ;
692
693         if(new_nbx*2 == theMesh->nbx)
```

```
694                 return(0) ;
695         else
696                 off_set = 1 ;
697
698         for(k=0;k<theMesh->nbx;k++) {
699                 for(l=0;l<theMesh->nby;l++) {
700
701                                         ks = 100 + (float) k ;
702                                         ls = (float) l ;
703
704                                         theblock->xnodes[0] = ks   ;
705                                         theblock->xnodes[1] = ks  ;
706                                         theblock->xnodes[2] = ks - 1. ;
707                                         theblock->xnodes[3] = ks - 1. ;
708
709                                 if(k == (theMesh->nbx-1)){
710                                         theblock->xnodes[0] = ks + 1. ;
711                                         theblock->xnodes[1] = ks + 1. ;
712                                         theblock->xnodes[2] = ks - 1. ;
713                                         theblock->xnodes[3] = ks - 1.  ;
714                                 }
715
716                                         theblock->ynodes[0] = 100. + 2.*ls -1. ;
717                                         theblock->ynodes[1] = 100. +2.*ls +1. ;
718                                         theblock->ynodes[2] = 100. +2.*ls +1. ;
719                                         theblock->ynodes[3] = 100. +2.*ls -1. ;
720
721                         map_mesh(theblock,211,l,k) ;
722                         theblock++ ;
723                 }
724         }
725
726         nodep = gp.N ;
727         for(i=0;i<N.nodes;i++){
728                 nodep->x[0] -= 100.0 ;
729                 nodep->x[1] -= 100.0 ;
730                 nodep = nodep->nextnp ;
731         }
732         free(theblock) ;
733         return(0) ;
734 }
735
736
737 int     test_analytical(nvar)
738 int     nvar ;
739 {
740         int             i, j, k, d_n ;
741         double eta, Q, neta, w, w_m_n, w_p_n ;
742         double          erf(), value, first_x ;
743         struct belement       *belp ;
744         struct node   *nodep ;
745
746
747         printf(" ||||||||||                         WARNING
748 |||||||||| \n\n") ;
749         printf(" |||| You are about to solve for the Analytical Solution||||\n")
750 ;
751         printf(" The rest of this program will treat p[2] as the \n") ;
752         printf(" Non-dimensional cummulative discharge !\n") ;
753
754
755         printf(" Please input the discharge and the fraction for the Line
756 source?\n->") ;
```

```
757         scanf(" %lf %lf", &Q, &w) ;
758         printf(" Discharge = %lf and w = %lf\n",Q,w) ;
759
760         d_n = -1000000 ;
761         belp = gp.B ;
762         for(i=0;i<N.belms;i++) {
763                 if((belp->nps[0]->n - belp->nps[1]->n) > d_n)
764                         d_n = (belp->nps[0]->n - belp->nps[1]->n) ;
765                 belp = belp->nextbelp ;
766         }
767
768         first_x = gp.iptrs[d_n]->x[0] - gp.iptrs[0]->x[0] ;
769
770         printf(" source at x = %lf\n",first_x) ;
771
772
773         nodep = gp.N ;
774         for(i=0;i<N.nodes;i++){
775
776                 if( i % 10 == 0.0 )
777                         printf(" Finished %d nodes\n",i) ;
778
779                 neta = 4.*nodep->p[5] * (nodep->x[0]-first_x)/ Q/Q ;
780
781
782                 if(neta <= 0.0)
783                         goto SKIP ;
784
785                 eta = sqrt(fabs(neta)) ;
786                 w_m_n = w - nodep->p[0] ;
787                 w_p_n = w + nodep->p[0] ;
788
789                 printf(" for node %d, neta = %lf, eta = %lf, w_m_n = %lf, w_p_n =
790 %lf\n",
791                                         i, neta, eta, w_m_n, w_p_n) ;
792 /*                      printf(" value = %lf \n",value) ;
793 */
794                 value = erf(w_m_n / eta) + erf(w_p_n / eta) ;
795                 for(j=1;j<3;j++) {
796                         value += ( erf((2*j + w_m_n) / eta) + erf((2*j + w_p_n)) /
797 eta ) ;
798                         value -= ( erf((2*j - w_p_n) / eta) + erf((2*j - w_m_n)) /
799 eta) ;
800                 }
801
802                 nodep->u[0] = value / 2. / w ;
803 SKIP :
804                 nodep = nodep->nextnp ;
805         }
806         return(0) ;
807 }
808
809 double          erf(x)
810 double          x ;
811 {
812         double          v, result, fact = 1.0, delta ;
813
814         delta = 0.0001 ;
815
816         v = 0.0 ;
817         if(x < 0.0){
818                 fact = -1.0 ;
819                 x *= fact ;
```

```
820         }
821
822         result = 0.0 ;
823         while( v <= x ){
824                 result +=2/1.772453851*delta*(exp(-v*v)+exp(-
825   (v+delta)*(v+delta)))/2. ;
826                 delta  *= 1.001 ;
827                 v += delta ;
828         }
829         return(result*fact) ;
830   }
831
832
833   int     special_out(nin)
834   int nin;
835   {
836         int                     err,i,dir ;
837         FILE                    *fptr, *put_fptr() ;
838         struct node     *np ;
839         struct element          *elp ;
840         float           val;
841         char            l_dir;
842
843         if ( (fptr = put_fptr(1)) != NULL){
844                 printf("Please input the direction along which the output is
845   desired!");
846
847                 printf("\n >");
848                 scanf(" %c",&l_dir);
849                 if (l_dir == 'x'){
850                         printf("\n Input the y coordinate.");
851                         dir = 1;
852                 }
853                 else {
854                         printf("\n Input the x coordinate.");
855                         dir = 0;
856                 }
857                 printf("\n >");
858                 scanf(" %f",&val);
859                 np = gp.N;
860                 for (i=0;i<N.nodes;i++){
861                         if (np->x[dir] == val){
862                                 if (nin == 1){
863                                         printf(" %f\t%f\t%f\n",np->x[0], np-
864   >x[1],np->u[0]);
865                                 }
866                                 fprintf(fptr," %f\t%f\t%20.14g\n",np->x[0], np-
867   >x[1],np->u[0]);
868                         }
869                         np = np->nextnp;
870                 }
871                 if (nin == 1 || nin == 2){
872                         fclose(fptr);
873                         return(1);
874                 }
875                 return(1);
876         }
877         else {
878                 printf("Couldn't open the specified file");
879                 return(-1);
880         }
881   }
882
```

```
1   #include "fe.h"
2
3
4   extern struct control       N ;
5
6   int             QU1DPick(elmntp,theElp,ntfp,dirp)
7   struct element      *elmntp, *theElp ;
8   int             *ntfp, *dirp ;
9
10  {
11          int             j;
12          double          q01 ;
13          struct node     *np[MNSF] ;
14
15          theElp->gtype = 111 ;
16          *ntfp = 2;
17          for(j=0;j<elmntp->nnds;j++)
18                  np[j] = elmntp->nps[j] ;
19
20          for(j=0;j<N.params;j++)
21                  theElp->p[j] = elmntp->p[j] ;
22
23  /*
24          q01 = np[0]->p[0] * (np[1]->x[0]-np[0]->x[0])
25                  + np[0]->p[1] *(np[1]->x[1]-np[0]->x[1]) ;
26
27  */
28          q01 = (np[0]->p[0]+np[1]->p[0])/2*(np[1]->x[0]-np[0]->x[0])
29                          + (np[0]->p[1]+np[1]->p[1])/2*(np[1]->x[1]-np[0]->x[1]) ;
30
31          if (q01 >= 0.0) {
32                  if(elmntp->nps[2] == NULL ){
33                          theElp->vtype = theElp->gtype = 111 ;
34                          theElp->nnds = 2 ;
35                          theElp->nps[0] = elmntp->nps[0] ;
36                          theElp->nps[1] = elmntp->nps[1] ;
37                          return(0) ;
38                  }
39                  else {
40                          theElp->vtype = 129 ;
41                          theElp->nnds = 3 ;
42                          theElp->nps[0] = elmntp->nps[0] ;
43                          theElp->nps[1] = elmntp->nps[1] ;
44                          theElp->nps[2] = elmntp->nps[2] ;
45                          return(1) ;
46                  }
47          }
48          else {
49                  if(elmntp->nps[3] == NULL ){
50                          theElp->vtype = theElp->gtype = 111 ;
51                          theElp->nnds = 2 ;
52                          theElp->nps[0] = elmntp->nps[0] ;
53                          theElp->nps[1] = elmntp->nps[1] ;
54                          return(0) ;
55                  }
56                  else  {
57                          theElp->vtype = 128 ;
58                          theElp->gtype = 111 ;
59                          theElp->nnds = 3 ;
60                          theElp->nps[0] = elmntp->nps[0] ;
61                          theElp->nps[1] = elmntp->nps[1] ;
62                          theElp->nps[2] = elmntp->nps[3] ;
63                          return(1) ;
```

```
64                      }
65              }
66      }
67
68
69      int             CU1DPick(elmntp,theElp,ntfp,dirp)
70      struct element          *elmntp, *theElp ;
71      int             *ntfp, *dirp ;
72
73      {
74              int             j;
75              double          q01 ;
76              struct node     *np[MNSF] ;
77
78              theElp->gtype = 111 ;
79              *ntfp = 2;
80              for(j=0;j<elmntp->nnds;j++)
81                      np[j] = elmntp->nps[j] ;
82
83              for(j=0;j<N.params;j++)
84                      theElp->p[j] = elmntp->p[j] ;
85
86              q01 = (np[0]->p[0]+np[1]->p[0])/2*(np[1]->x[0]-np[0]->x[0])
87                      + (np[0]->p[1]+np[1]->p[1])/2*(np[1]->x[1]-np[0]->x[1]) ;
88
89              if (q01 >= 0.0) {
90                      if((elmntp->nps[2] == NULL) || (elmntp->nps[3] == NULL) ){
91                              theElp->vtype = theElp->gtype = 111 ;
92                              theElp->nnds = 2 ;
93                              theElp->nps[0] = elmntp->nps[0] ;
94                              theElp->nps[1] = elmntp->nps[1] ;
95                              return(0) ;
96                      }
97                      else {
98                              theElp->vtype = 139 ;
99                              theElp->gtype = 111 ;
100                             *ntfp = 2;
101                             theElp->nnds = 4 ;
102                             theElp->nps[0] = elmntp->nps[0] ;
103                             theElp->nps[1] = elmntp->nps[1] ;
104                             theElp->nps[2] = elmntp->nps[2] ;
105                             theElp->nps[3] = elmntp->nps[3] ;
106                             return(1) ;
107                     }
108             }
109             else {
110                     if((elmntp->nps[4] == NULL) || (elmntp->nps[5] == NULL) ){
111                             theElp->vtype = theElp->gtype = 111 ;
112                             theElp->nnds = 2 ;
113                             theElp->nps[0] = elmntp->nps[0] ;
114                             theElp->nps[1] = elmntp->nps[1] ;
115                             return(0) ;
116                     }
117                     else  {
118                             theElp->vtype = 138 ;
119                             theElp->gtype = 111 ;
120                             *ntfp = 2;
121                             theElp->nnds = 4 ;
122                             theElp->nps[0] = elmntp->nps[0] ;
123                             theElp->nps[1] = elmntp->nps[1] ;
124                             theElp->nps[2] = elmntp->nps[4] ;
125                             theElp->nps[3] = elmntp->nps[5] ;
126                             return(1) ;
```

```
127                     }
128             }
129     /*      return(-1) ;  */
130     }
131
132
133     int             QUelPick(elmntp,theElp,ntfp,dirp)
134     struct element          *elmntp, *theElp ;
135     int             *ntfp, *dirp ;
136
137     {
138             int             j;
139             double          q01, q12, q23, q30 ;
140             struct node     *np[MNSF] ;
141
142             theElp->gtype = 219 ;
143             *ntfp = 4;
144             for(j=0;j<elmntp->nnds;j++)
145                     np[j] = elmntp->nps[j] ;
146
147             for(j=0;j<N.params;j++)
148                     theElp->p[j] = elmntp->p[j] ;
149
150             q01 = (np[0]->p[0]+np[1]->p[0])/2*(np[0]->x[1]-np[1]->x[1])
151                             - (np[0]->p[1]+np[1]->p[1])/2*(np[0]->x[0]-np[1]->x[0]) ;
152             q12 = (np[1]->p[0]+np[2]->p[0])/2*(np[1]->x[1]-np[2]->x[1])
153                             - (np[1]->p[1]+np[2]->p[1])/2*(np[1]->x[0]-np[2]->x[0]) ;
154             q23 = (np[2]->p[0]+np[3]->p[0])/2*(np[2]->x[1]-np[3]->x[1])
155                             - (np[2]->p[1]+np[3]->p[1])/2*(np[2]->x[0]-np[3]->x[0]) ;
156             q30 = (np[3]->p[0]+np[0]->p[0])/2*(np[3]->x[1]-np[0]->x[1])
157                             - (np[3]->p[1]+np[0]->p[1])/2*(np[3]->x[0]-np[0]->x[0]) ;
158
159             if((q23>=q01) && (q30>=q12)) {
160                     if((elmntp->nps[14] == NULL) || (elmntp->nps[15] == NULL) ){
161                             if((elmntp->nps[11] == NULL) || (elmntp->nps[12] == NULL)
162     ){
163                                     theElp->vtype = theElp->gtype = 211 ;
164                                     theElp->nnds = 4 ;
165                                     theElp->nps[0] = elmntp->nps[0] ;
166                                     theElp->nps[1] = elmntp->nps[1] ;
167                                     theElp->nps[2] = elmntp->nps[2] ;
168                                     theElp->nps[3] = elmntp->nps[3] ;
169                                     return(0) ;
170                             }
171                             else {
172                                     theElp->vtype = 228 ;
173                                     theElp->nnds = 6 ;
174                                     theElp->nps[0] = elmntp->nps[0] ;
175                                     theElp->nps[1] = elmntp->nps[1] ;
176                                     theElp->nps[2] = elmntp->nps[2] ;
177                                     theElp->nps[3] = elmntp->nps[3] ;
178                                     theElp->nps[4] = elmntp->nps[11] ;
179                                     theElp->nps[5] = elmntp->nps[12] ;
180                                     return(1) ;
181                             }
182                     }
183                     else {
184                             if((elmntp->nps[11] == NULL) || (elmntp->nps[12] == NULL)
185     ){
186                                     theElp->vtype = 228 ;
187                                     theElp->nnds = 6 ;
188                                     theElp->nps[0] = elmntp->nps[1] ;
189                                     theElp->nps[1] = elmntp->nps[2] ;
```

```
190                             theElp->nps[2] = elmntp->nps[3] ;
191                             theElp->nps[3] = elmntp->nps[0] ;
192                             theElp->nps[4] = elmntp->nps[14] ;
193                             theElp->nps[5] = elmntp->nps[15] ;
194                             return(1) ;
195                     }
196                     else if (elmntp->nps[13] == NULL) {
197                             theElp->vtype = 224 ;
198                             theElp->nnds = 8 ;
199                             theElp->nps[0] = elmntp->nps[0] ;
200                             theElp->nps[1] = elmntp->nps[1] ;
201                             theElp->nps[2] = elmntp->nps[2] ;
202                             theElp->nps[3] = elmntp->nps[3] ;
203                             theElp->nps[4] = elmntp->nps[11] ;
204                             theElp->nps[5] = elmntp->nps[12] ;
205                             theElp->nps[7] = elmntp->nps[14] ;
206                             theElp->nps[8] = elmntp->nps[15] ;
207                             return(2) ;
208                     }
209                     else {
210                             theElp->vtype = 229 ;
211                             theElp->nnds = 9 ;
212                             theElp->nps[0] = elmntp->nps[0] ;
213                             theElp->nps[1] = elmntp->nps[1] ;
214                             theElp->nps[2] = elmntp->nps[2] ;
215                             theElp->nps[3] = elmntp->nps[3] ;
216                             theElp->nps[4] = elmntp->nps[11] ;
217                             theElp->nps[5] = elmntp->nps[12] ;
218                             theElp->nps[6] = elmntp->nps[13] ;
219                             theElp->nps[7] = elmntp->nps[14] ;
220                             theElp->nps[8] = elmntp->nps[15] ;
221                             return(3) ;
222                     }
223             }
224     }
225     if((q23>=q01) && (q30<q12)) {
226             if((elmntp->nps[11] == NULL) || (elmntp->nps[12] == NULL) ){
227                     if((elmntp->nps[8] == NULL) || (elmntp->nps[9] == NULL) ){
228                             theElp->vtype = 211 ;
229                             theElp->nnds = 4 ;
230                             theElp->nps[0] = elmntp->nps[0] ;
231                             theElp->nps[1] = elmntp->nps[1] ;
232                             theElp->nps[2] = elmntp->nps[2] ;
233                             theElp->nps[3] = elmntp->nps[3] ;
234                             return(0) ;
235                     }
236                     else {
237                             theElp->vtype = 228 ;
238                             theElp->nnds = 6 ;
239                             theElp->nps[0] = elmntp->nps[3] ;
240                             theElp->nps[1] = elmntp->nps[0] ;
241                             theElp->nps[2] = elmntp->nps[1] ;
242                             theElp->nps[3] = elmntp->nps[2] ;
243                             theElp->nps[4] = elmntp->nps[8] ;
244                             theElp->nps[5] = elmntp->nps[9] ;
245                             return(1) ;
246                     }
247             }
248             else {
249                     if((elmntp->nps[8] == NULL) || (elmntp->nps[9] == NULL) ){
250                             theElp->vtype = 228 ;
251                             theElp->nnds = 6 ;
252                             theElp->nps[0] = elmntp->nps[0] ;
```

```
253                             theElp->nps[1] = elmntp->nps[1] ;
254                             theElp->nps[2] = elmntp->nps[2] ;
255                             theElp->nps[3] = elmntp->nps[3] ;
256                             theElp->nps[4] = elmntp->nps[11] ;
257                             theElp->nps[5] = elmntp->nps[12] ;
258                             return(1) ;
259                     }
260                     else if (elmntp->nps[10] == NULL) {
261                             theElp->vtype = 224 ;
262                             theElp->nnds = 8 ;
263                             theElp->nps[0] = elmntp->nps[3] ;
264                             theElp->nps[1] = elmntp->nps[0] ;
265                             theElp->nps[2] = elmntp->nps[1] ;
266                             theElp->nps[3] = elmntp->nps[2] ;
267                             theElp->nps[4] = elmntp->nps[8] ;
268                             theElp->nps[5] = elmntp->nps[9] ;
269                             theElp->nps[7] = elmntp->nps[11] ;
270                             theElp->nps[8] = elmntp->nps[12] ;
271                             return(2) ;
272                     }
273                     else {
274                             theElp->vtype = 229 ;
275                             theElp->nnds = 9 ;
276                             theElp->nps[0] = elmntp->nps[3] ;
277                             theElp->nps[1] = elmntp->nps[0] ;
278                             theElp->nps[2] = elmntp->nps[1] ;
279                             theElp->nps[3] = elmntp->nps[2] ;
280                             theElp->nps[4] = elmntp->nps[8] ;
281                             theElp->nps[5] = elmntp->nps[9] ;
282                             theElp->nps[6] = elmntp->nps[10] ;
283                             theElp->nps[7] = elmntp->nps[11] ;
284                             theElp->nps[8] = elmntp->nps[12] ;
285                             return(3) ;
286                     }
287             }
288     }
289     if((q23<q01) && (q30<q12)) {
290             if((elmntp->nps[8] == NULL) || (elmntp->nps[9] == NULL) ){
291                     if((elmntp->nps[5] == NULL) || (elmntp->nps[6] == NULL) ){
292                             theElp->vtype = 211 ;
293                             theElp->nnds = 4 ;
294                             theElp->nps[0] = elmntp->nps[0] ;
295                             theElp->nps[1] = elmntp->nps[1] ;
296                             theElp->nps[2] = elmntp->nps[2] ;
297                             theElp->nps[3] = elmntp->nps[3] ;
298                             return(0) ;
299                     }
300                     else {
301                             theElp->vtype = 228 ;
302                             theElp->nnds = 6 ;
303                             theElp->nps[0] = elmntp->nps[2] ;
304                             theElp->nps[1] = elmntp->nps[3] ;
305                             theElp->nps[2] = elmntp->nps[0] ;
306                             theElp->nps[3] = elmntp->nps[1] ;
307                             theElp->nps[4] = elmntp->nps[5] ;
308                             theElp->nps[5] = elmntp->nps[6] ;
309                             return(1) ;
310                     }
311             }
312             else {
313                     if((elmntp->nps[5] == NULL) || (elmntp->nps[6] == NULL) ){
314                             theElp->vtype = 228 ;
315                             theElp->nnds = 6 ;
```

```
316                             theElp->nps[0] = elmntp->nps[3] ;
317                             theElp->nps[1] = elmntp->nps[0] ;
318                             theElp->nps[2] = elmntp->nps[1] ;
319                             theElp->nps[3] = elmntp->nps[2] ;
320                             theElp->nps[4] = elmntp->nps[8] ;
321                             theElp->nps[5] = elmntp->nps[9] ;
322                             return(1) ;
323                     }
324                     else if (elmntp->nps[7] == NULL) {
325                             theElp->vtype = 224 ;
326                             theElp->nnds = 8 ;
327                             theElp->nps[0] = elmntp->nps[2] ;
328                             theElp->nps[1] = elmntp->nps[3] ;
329                             theElp->nps[2] = elmntp->nps[0] ;
330                             theElp->nps[3] = elmntp->nps[1] ;
331                             theElp->nps[4] = elmntp->nps[5] ;
332                             theElp->nps[5] = elmntp->nps[6] ;
333                             theElp->nps[6] = elmntp->nps[8] ;
334                             theElp->nps[7] = elmntp->nps[9] ;
335                             return(2) ;
336                     }
337                     else {
338                             theElp->vtype = 229 ;
339                             theElp->nnds = 9 ;
340                             theElp->nps[0] = elmntp->nps[2] ;
341                             theElp->nps[1] = elmntp->nps[3] ;
342                             theElp->nps[2] = elmntp->nps[0] ;
343                             theElp->nps[3] = elmntp->nps[1] ;
344                             theElp->nps[4] = elmntp->nps[5] ;
345                             theElp->nps[5] = elmntp->nps[6] ;
346                             theElp->nps[6] = elmntp->nps[7] ;
347                             theElp->nps[7] = elmntp->nps[8] ;
348                             theElp->nps[8] = elmntp->nps[9] ;
349                             return(3) ;
350                     }
351             }
352     }
353     if((q23<q01) && (q30>=q12)) {
354             if((elmntp->nps[5] == NULL) || (elmntp->nps[6] == NULL) ){
355                     if((elmntp->nps[14] == NULL) || (elmntp->nps[15] == NULL)
356 ){
357                             theElp->vtype = 211 ;
358                             theElp->nnds = 4 ;
359                             theElp->nps[0] = elmntp->nps[0] ;
360                             theElp->nps[1] = elmntp->nps[1] ;
361                             theElp->nps[2] = elmntp->nps[2] ;
362                             theElp->nps[3] = elmntp->nps[3] ;
363                             return(0) ;
364                     }
365                     else {
366                             theElp->vtype = 228 ;
367                             theElp->nnds = 6 ;
368                             theElp->nps[0] = elmntp->nps[1] ;
369                             theElp->nps[1] = elmntp->nps[2] ;
370                             theElp->nps[2] = elmntp->nps[3] ;
371                             theElp->nps[3] = elmntp->nps[0] ;
372                             theElp->nps[4] = elmntp->nps[14] ;
373                             theElp->nps[5] = elmntp->nps[15] ;
374                             return(1) ;
375                     }
376             }
377             else {
```

```
                        if((elmntp->nps[14] == NULL) || (elmntp->nps[15] == NULL)
){
                                theElp->vtype = 228 ;
                                theElp->nnds = 6 ;
                                theElp->nps[0] = elmntp->nps[2] ;
                                theElp->nps[1] = elmntp->nps[3] ;
                                theElp->nps[2] = elmntp->nps[0] ;
                                theElp->nps[3] = elmntp->nps[1] ;
                                theElp->nps[4] = elmntp->nps[5] ;
                                theElp->nps[5] = elmntp->nps[6] ;
                                return(1) ;
                        }
                        else if (elmntp->nps[4] == NULL) {
                                theElp->vtype = 224 ;
                                theElp->nnds = 8 ;
                                theElp->nps[0] = elmntp->nps[1] ;
                                theElp->nps[1] = elmntp->nps[2] ;
                                theElp->nps[2] = elmntp->nps[3] ;
                                theElp->nps[3] = elmntp->nps[0] ;
                                theElp->nps[4] = elmntp->nps[14] ;
                                theElp->nps[5] = elmntp->nps[15] ;
                                theElp->nps[6] = elmntp->nps[5] ;
                                theElp->nps[7] = elmntp->nps[6] ;
                                return(2) ;
                        }
                        else {
                                theElp->vtype = 229 ;
                                theElp->nnds = 9 ;
                                theElp->nps[0] = elmntp->nps[1] ;
                                theElp->nps[1] = elmntp->nps[2] ;
                                theElp->nps[2] = elmntp->nps[3] ;
                                theElp->nps[3] = elmntp->nps[0] ;
                                theElp->nps[4] = elmntp->nps[14] ;
                                theElp->nps[5] = elmntp->nps[15] ;
                                theElp->nps[6] = elmntp->nps[4] ;
                                theElp->nps[7] = elmntp->nps[5] ;
                                theElp->nps[8] = elmntp->nps[6] ;
                                return(3) ;
                        }
                }
        }
        return(-1) ;
}

int             CU_Stream_Pick(elmntp,theElp,ntfp,dirp)
struct          element         *elmntp, *theElp ;
int             *ntfp, *dirp ;
{
                theElp->vtype = 233 ;
                        theElp->nnds = 8 ;
                        theElp->nps[0] = elmntp->nps[0] ;
                        theElp->nps[1] = elmntp->nps[1] ;
                        theElp->nps[2] = elmntp->nps[2] ;
                        theElp->nps[3] = elmntp->nps[3] ;
                        theElp->nps[4] = elmntp->nps[4] ;
                        theElp->nps[5] = elmntp->nps[5] ;
                        theElp->nps[6] = elmntp->nps[6] ;
                        theElp->nps[7] = elmntp->nps[7] ;

                if (elmntp->nps[5] == NULL || elmntp->nps[6] == NULL){
                        theElp->vtype = 230;
                        theElp->nnds = 6 ;
                        theElp->nps[0] = elmntp->nps[0] ;
```

```
441                     theElp->nps[1] = elmntp->nps[1] ;
442                     theElp->nps[2] = elmntp->nps[2] ;
443                     theElp->nps[3] = elmntp->nps[3] ;
444                     theElp->nps[4] = elmntp->nps[4] ;
445                     theElp->nps[5] = elmntp->nps[7] ;
446             }
447
448             if(elmntp->nps[4] == NULL || elmntp->nps[7] == NULL){
449                     theElp->vtype = 211;
450                     theElp->nnds = 4 ;
451                     theElp->nps[0] = elmntp->nps[0] ;
452                     theElp->nps[1] = elmntp->nps[1] ;
453                     theElp->nps[2] = elmntp->nps[2] ;
454                     theElp->nps[3] = elmntp->nps[3] ;
455             }
456
457             return(-1) ;
458     }
459
460     int             CU2DPick(elmntp,theElp,ntfp,dirp)
461     struct element          *elmntp, *theElp ;
462     int             *ntfp, *dirp ;
463
464     {
465             int     j;
466             double q01, q12, q23, q30 ;
467             struct node   *np[MNSF] ;
468             int node_hands[MNSF] ;
469
470             theElp->gtype = 211 ;
471             *ntfp = 4;
472             for(j=0;j<elmntp->nnds;j++)
473                     np[j] = elmntp->nps[j] ;
474
475             for(j=0;j<N.params;j++)
476                     theElp->p[j] = elmntp->p[j] ;
477
478             q01 = (np[0]->p[0]+np[1]->p[0])/2*(np[0]->x[1]-np[1]->x[1])
479                             - (np[0]->p[1]+np[1]->p[1])/2*(np[0]->x[0]-np[1]->x[0]) ;
480             q12 = (np[1]->p[0]+np[2]->p[0])/2*(np[1]->x[1]-np[2]->x[1])
481                             - (np[1]->p[1]+np[2]->p[1])/2*(np[1]->x[0]-np[2]->x[0]) ;
482             q23 = (np[2]->p[0]+np[3]->p[0])/2*(np[2]->x[1]-np[3]->x[1])
483                             - (np[2]->p[1]+np[3]->p[1])/2*(np[2]->x[0]-np[3]->x[0]) ;
484             q30 = (np[3]->p[0]+np[0]->p[0])/2*(np[3]->x[1]-np[0]->x[1])
485                             - (np[3]->p[1]+np[0]->p[1])/2*(np[3]->x[0]-np[0]->x[0]) ;
486
487             if((q23>=q01) && (q30>=q12)) {
488                     node_hands[0] = 0 ;
489                     node_hands[1] = 1 ;
490                     node_hands[2] = 2 ;
491                     node_hands[3] = 3 ;
492                     node_hands[4] = 11 ;
493                     node_hands[5] = 28 ;
494                     node_hands[6] = 29 ;
495                     node_hands[7] = 12 ;
496                     node_hands[8] = 30 ;
497                     node_hands[9] = 13 ;
498                     node_hands[10] = 14 ;
499                     node_hands[11] = 15 ;
500                     node_hands[12] = 31 ;
501                     node_hands[13] = 32 ;
502                     node_hands[14] = 33 ;
503                     node_hands[15] = 34 ;
```

```
504         }
505         if((q23>=q01) && (q30<q12)) {
506                 node_hands[0] = 3 ;
507                 node_hands[1] = 0 ;
508                 node_hands[2] = 1 ;
509                 node_hands[3] = 2 ;
510                 node_hands[4] = 8 ;
511                 node_hands[5] = 23 ;
512                 node_hands[6] = 24 ;
513                 node_hands[7] = 9 ;
514                 node_hands[8] = 25 ;
515                 node_hands[9] = 10 ;
516                 node_hands[10] = 11 ;
517                 node_hands[11] = 12 ;
518                 node_hands[12] = 26 ;
519                 node_hands[13] = 27 ;
520                 node_hands[14] = 28 ;
521                 node_hands[15] = 29 ;
522         }
523         if((q23<q01) && (q30<q12)) {
524                 node_hands[0] = 2 ;
525                 node_hands[1] = 3 ;
526                 node_hands[2] = 0 ;
527                 node_hands[3] = 1 ;
528                 node_hands[4] = 5 ;
529                 node_hands[5] = 18 ;
530                 node_hands[6] = 19 ;
531                 node_hands[7] = 6 ;
532                 node_hands[8] = 20 ;
533                 node_hands[9] = 7 ;
534                 node_hands[10] = 8 ;
535                 node_hands[11] = 9 ;
536                 node_hands[12] = 21 ;
537                 node_hands[13] = 22 ;
538                 node_hands[14] = 23 ;
539                 node_hands[15] = 24 ;
540         }
541         if((q23<q01) && (q30>=q12)) {
542                 node_hands[0] = 1 ;
543                 node_hands[1] = 2 ;
544                 node_hands[2] = 3 ;
545                 node_hands[3] = 0 ;
546                 node_hands[4] = 14 ;
547                 node_hands[5] = 33 ;
548                 node_hands[6] = 34 ;
549                 node_hands[7] = 15 ;
550                 node_hands[8] = 35 ;
551                 node_hands[9] = 4 ;
552                 node_hands[10] = 5 ;
553                 node_hands[11] = 6 ;
554                 node_hands[12] = 16 ;
555                 node_hands[13] = 17 ;
556                 node_hands[14] = 18 ;
557                 node_hands[15] = 19 ;
558         }
559         reall_nodes_CU2D( elmntp, theElp, node_hands) ;
560
561         return(-1) ;
562 }
563
564
565 int             reall_nodes_CU2D(elmntp, theElp, n_h)
566 struct element      *elmntp, *theElp ;
```

```
567   int           n_h[MNSF] ;
568   {
569
570                 if((elmntp->nps[n_h[12]] == NULL)){
571                       theElp->vtype = 238 ;
572                       theElp->nnds = 15 ;
573                       theElp->nps[0] = elmntp->nps[n_h[0]] ;
574                       theElp->nps[1] = elmntp->nps[n_h[1]] ;
575                       theElp->nps[2] = elmntp->nps[n_h[2]] ;
576                       theElp->nps[3] = elmntp->nps[n_h[3]] ;
577                       theElp->nps[4] = elmntp->nps[n_h[4]] ;
578                       theElp->nps[5] = elmntp->nps[n_h[5]] ;
579                       theElp->nps[6] = elmntp->nps[n_h[6]] ;
580                       theElp->nps[7] = elmntp->nps[n_h[7]] ;
581                       theElp->nps[8] = elmntp->nps[n_h[8]] ;
582                       theElp->nps[9] = elmntp->nps[n_h[9]] ;
583                       theElp->nps[10] = elmntp->nps[n_h[10]] ;
584                       theElp->nps[11] = elmntp->nps[n_h[13]] ;
585                       theElp->nps[12] = elmntp->nps[n_h[14]] ;
586                       theElp->nps[13] = elmntp->nps[n_h[15]] ;
587                       theElp->nps[14] = elmntp->nps[n_h[11]] ;
588
589                 }
590                 else {
591                       theElp->vtype = 239 ;
592                       theElp->nnds = 16 ;
593                       theElp->nps[0] = elmntp->nps[n_h[0]] ;
594                       theElp->nps[1] = elmntp->nps[n_h[1]] ;
595                       theElp->nps[2] = elmntp->nps[n_h[2]] ;
596                       theElp->nps[3] = elmntp->nps[n_h[3]] ;
597                       theElp->nps[4] = elmntp->nps[n_h[4]] ;
598                       theElp->nps[5] = elmntp->nps[n_h[5]] ;
599                       theElp->nps[6] = elmntp->nps[n_h[6]] ;
600                       theElp->nps[7] = elmntp->nps[n_h[7]] ;
601                       theElp->nps[8] = elmntp->nps[n_h[8]] ;
602                       theElp->nps[9] = elmntp->nps[n_h[9]] ;
603                       theElp->nps[10] = elmntp->nps[n_h[10]] ;
604                       theElp->nps[11] = elmntp->nps[n_h[12]] ;
605                       theElp->nps[12] = elmntp->nps[n_h[13]] ;
606                       theElp->nps[13] = elmntp->nps[n_h[14]] ;
607                       theElp->nps[14] = elmntp->nps[n_h[15]] ;
608                       theElp->nps[15] = elmntp->nps[n_h[11]] ;
609
610                 }
611                 if ((elmntp->nps[n_h[13]] == NULL)){
612                       theElp->vtype = 2371 ;
613                       theElp->nnds = 14 ;
614                       theElp->nps[0] = elmntp->nps[n_h[0]] ;
615                       theElp->nps[1] = elmntp->nps[n_h[1]] ;
616                       theElp->nps[2] = elmntp->nps[n_h[2]] ;
617                       theElp->nps[3] = elmntp->nps[n_h[3]] ;
618                       theElp->nps[4] = elmntp->nps[n_h[4]] ;
619                       theElp->nps[5] = elmntp->nps[n_h[5]] ;
620                       theElp->nps[6] = elmntp->nps[n_h[6]] ;
621                       theElp->nps[7] = elmntp->nps[n_h[7]] ;
622                       theElp->nps[8] = elmntp->nps[n_h[8]] ;
623                       theElp->nps[9] = elmntp->nps[n_h[9]] ;
624                       theElp->nps[10] = elmntp->nps[n_h[10]] ;
625                       theElp->nps[11] = elmntp->nps[n_h[14]] ;
626                       theElp->nps[12] = elmntp->nps[n_h[15]] ;
627                       theElp->nps[13] = elmntp->nps[n_h[11]] ;
628
629                 }
```

```
630                 if ((elmntp->nps[n_h[8]] == NULL)){
631                         theElp->vtype = 2372 ;
632                         theElp->nnds = 14 ;
633                         theElp->nps[0] = elmntp->nps[n_h[0]] ;
634                         theElp->nps[1] = elmntp->nps[n_h[1]] ;
635                         theElp->nps[2] = elmntp->nps[n_h[2]] ;
636                         theElp->nps[3] = elmntp->nps[n_h[3]] ;
637                         theElp->nps[4] = elmntp->nps[n_h[11]] ;
638                         theElp->nps[5] = elmntp->nps[n_h[15]] ;
639                         theElp->nps[6] = elmntp->nps[n_h[14]] ;
640                         theElp->nps[7] = elmntp->nps[n_h[10]] ;
641                         theElp->nps[8] = elmntp->nps[n_h[13]] ;
642                         theElp->nps[9] = elmntp->nps[n_h[9]] ;
643                         theElp->nps[10] = elmntp->nps[n_h[7]] ;
644                         theElp->nps[11] = elmntp->nps[n_h[6]] ;
645                         theElp->nps[12] = elmntp->nps[n_h[5]] ;
646                         theElp->nps[13] = elmntp->nps[n_h[4]] ;
647
648                 }
649                 if ((elmntp->nps[n_h[14]] == NULL) || (elmntp->nps[n_h[15]] ==
650 NULL)){
651                         theElp->vtype = 2361;
652                         theElp->nnds = 12;
653                         theElp->nps[0] = elmntp->nps[n_h[0]] ;
654                         theElp->nps[1] = elmntp->nps[n_h[1]] ;
655                         theElp->nps[2] = elmntp->nps[n_h[2]] ;
656                         theElp->nps[3] = elmntp->nps[n_h[3]] ;
657                         theElp->nps[4] = elmntp->nps[n_h[4]] ;
658                         theElp->nps[5] = elmntp->nps[n_h[5]] ;
659                         theElp->nps[6] = elmntp->nps[n_h[6]] ;
660                         theElp->nps[7] = elmntp->nps[n_h[7]] ;
661                         theElp->nps[8] = elmntp->nps[n_h[8]] ;
662                         theElp->nps[9] = elmntp->nps[n_h[9]] ;
663                         theElp->nps[10] = elmntp->nps[n_h[10]] ;
664                         theElp->nps[11] = elmntp->nps[n_h[11]] ;
665
666                 }
667                 if ((elmntp->nps[n_h[5]] == NULL) || (elmntp->nps[n_h[6]] ==
668 NULL)){
669                         theElp->vtype = 2363;
670                         theElp->nnds = 12;
671                         theElp->nps[0] = elmntp->nps[n_h[0]] ;
672                         theElp->nps[1] = elmntp->nps[n_h[1]] ;
673                         theElp->nps[2] = elmntp->nps[n_h[2]] ;
674                         theElp->nps[3] = elmntp->nps[n_h[3]] ;
675                         theElp->nps[4] = elmntp->nps[n_h[11]] ;
676                         theElp->nps[5] = elmntp->nps[n_h[15]] ;
677                         theElp->nps[6] = elmntp->nps[n_h[14]] ;
678                         theElp->nps[7] = elmntp->nps[n_h[10]] ;
679                         theElp->nps[8] = elmntp->nps[n_h[13]] ;
680                         theElp->nps[9] = elmntp->nps[n_h[9]] ;
681                         theElp->nps[10] = elmntp->nps[n_h[7]] ;
682                         theElp->nps[11] = elmntp->nps[n_h[4]] ;
683
684                 }
685                 if ((elmntp->nps[n_h[8]] == NULL) && ((elmntp->nps[n_h[14]] ==
686 NULL)
687
688         || (elmntp->nps[n_h[15]] == NULL))){
689                         theElp->vtype = 2351;
690                         theElp->nnds = 11 ;
691                         theElp->nps[0] = elmntp->nps[n_h[0]] ;
692                         theElp->nps[1] = elmntp->nps[n_h[1]] ;
```

```
693                     theElp->nps[2] = elmntp->nps[n_h[2]] ;
694                     theElp->nps[3] = elmntp->nps[n_h[3]] ;
695                     theElp->nps[4] = elmntp->nps[n_h[4]] ;
696                     theElp->nps[5] = elmntp->nps[n_h[5]] ;
697                     theElp->nps[6] = elmntp->nps[n_h[6]] ;
698                     theElp->nps[7] = elmntp->nps[n_h[7]] ;
699                     theElp->nps[8] = elmntp->nps[n_h[9]] ;
700                     theElp->nps[9] = elmntp->nps[n_h[10]] ;
701                     theElp->nps[10] = elmntp->nps[n_h[11]] ;
702
703             }
704             if ((elmntp->nps[n_h[13]] == NULL) && ((elmntp->nps[n_h[5]] ==
705 NULL)
706                             || (elmntp->nps[n_h[6]] == NULL))){
707                     theElp->vtype = 2353;
708                     theElp->nnds = 11 ;
709                     theElp->nps[0] = elmntp->nps[n_h[0]] ;
710                     theElp->nps[1] = elmntp->nps[n_h[1]] ;
711                     theElp->nps[2] = elmntp->nps[n_h[2]] ;
712                     theElp->nps[3] = elmntp->nps[n_h[3]] ;
713                     theElp->nps[4] = elmntp->nps[n_h[11]] ;
714                     theElp->nps[5] = elmntp->nps[n_h[15]] ;
715                     theElp->nps[6] = elmntp->nps[n_h[14]] ;
716                     theElp->nps[7] = elmntp->nps[n_h[10]] ;
717                     theElp->nps[8] = elmntp->nps[n_h[9]] ;
718                     theElp->nps[9] = elmntp->nps[n_h[7]] ;
719                     theElp->nps[10] = elmntp->nps[n_h[4]] ;
720             }
721             if((elmntp->nps[n_h[8]] == NULL) && (elmntp->nps[n_h[13]] ==
722 NULL)){
723                     theElp->vtype = 2362 ;
724                     theElp->nnds = 13 ;
725                     theElp->nps[0] = elmntp->nps[n_h[0]] ;
726                     theElp->nps[1] = elmntp->nps[n_h[1]] ;
727                     theElp->nps[2] = elmntp->nps[n_h[2]] ;
728                     theElp->nps[3] = elmntp->nps[n_h[3]] ;
729                     theElp->nps[4] = elmntp->nps[n_h[4]] ;
730                     theElp->nps[5] = elmntp->nps[n_h[5]] ;
731                     theElp->nps[6] = elmntp->nps[n_h[6]] ;
732                     theElp->nps[7] = elmntp->nps[n_h[7]] ;
733                     theElp->nps[8] = elmntp->nps[n_h[9]] ;
734                     theElp->nps[9] = elmntp->nps[n_h[10]] ;
735                     theElp->nps[10] = elmntp->nps[n_h[14]] ;
736                     theElp->nps[11] = elmntp->nps[n_h[15]] ;
737                     theElp->nps[12] = elmntp->nps[n_h[11]] ;
738
739             }
740             if((elmntp->nps[n_h[9]] == NULL)){
741                     theElp->vtype = 2352 ;
742                     theElp->nnds = 12 ;
743                     theElp->nps[0] = elmntp->nps[n_h[0]] ;
744                     theElp->nps[1] = elmntp->nps[n_h[1]] ;
745                     theElp->nps[2] = elmntp->nps[n_h[2]] ;
746                     theElp->nps[3] = elmntp->nps[n_h[3]] ;
747                     theElp->nps[4] = elmntp->nps[n_h[4]] ;
748                     theElp->nps[5] = elmntp->nps[n_h[5]] ;
749                     theElp->nps[6] = elmntp->nps[n_h[6]] ;
750                     theElp->nps[7] = elmntp->nps[n_h[7]] ;
751                     theElp->nps[8] = elmntp->nps[n_h[10]] ;
752                     theElp->nps[9] = elmntp->nps[n_h[14]] ;
753                     theElp->nps[10] = elmntp->nps[n_h[15]] ;
754                     theElp->nps[11] = elmntp->nps[n_h[11]] ;
755             }
```

```
756
757                 else if(((elmntp->nps[n_h[6]] == NULL)    || (elmntp->nps[n_h[5]]
758     == NULL))
759                                 && ((elmntp->nps[n_h[14]] == NULL) || (elmntp-
760     >nps[n_h[15]] == NULL))){
761                         theElp->vtype = 232 ;
762                         theElp->nnds = 9 ;
763                         theElp->nps[0] = elmntp->nps[n_h[0]] ;
764                         theElp->nps[1] = elmntp->nps[n_h[1]] ;
765                         theElp->nps[2] = elmntp->nps[n_h[2]] ;
766                         theElp->nps[3] = elmntp->nps[n_h[3]] ;
767                         theElp->nps[4] = elmntp->nps[n_h[4]] ;
768                         theElp->nps[5] = elmntp->nps[n_h[7]] ;
769                         theElp->nps[6] = elmntp->nps[n_h[9]] ;
770                         theElp->nps[7] = elmntp->nps[n_h[10]] ;
771                         theElp->nps[8] = elmntp->nps[n_h[11]] ;
772
773                 }
774                 if(((elmntp->nps[n_h[14]] == NULL) || (elmntp->nps[n_h[15]] ==
775     NULL))
776                  && (elmntp->nps[n_h[9]] == NULL)){
777                         theElp->vtype = 2341 ;
778                         theElp->nnds = 10 ;
779                         theElp->nps[0] = elmntp->nps[n_h[0]] ;
780                         theElp->nps[1] = elmntp->nps[n_h[1]] ;
781                         theElp->nps[2] = elmntp->nps[n_h[2]] ;
782                         theElp->nps[3] = elmntp->nps[n_h[3]] ;
783                         theElp->nps[4] = elmntp->nps[n_h[4]] ;
784                         theElp->nps[5] = elmntp->nps[n_h[5]] ;
785                         theElp->nps[6] = elmntp->nps[n_h[6]] ;
786                         theElp->nps[7] = elmntp->nps[n_h[7]] ;
787                         theElp->nps[8] = elmntp->nps[n_h[10]] ;
788                         theElp->nps[9] = elmntp->nps[n_h[11]] ;
789
790                         if((elmntp->nps[n_h[6]] == NULL)    || (elmntp->nps[n_h[5]]
791     == NULL)){
792                                 theElp->vtype = 2311 ;
793                                 theElp->nnds = 8 ;
794                                 theElp->nps[0] = elmntp->nps[n_h[0]] ;
795                                 theElp->nps[1] = elmntp->nps[n_h[1]] ;
796                                 theElp->nps[2] = elmntp->nps[n_h[2]] ;
797                                 theElp->nps[3] = elmntp->nps[n_h[3]] ;
798                                 theElp->nps[4] = elmntp->nps[n_h[4]] ;
799                                 theElp->nps[5] = elmntp->nps[n_h[7]] ;
800                                 theElp->nps[6] = elmntp->nps[n_h[10]] ;
801                                 theElp->nps[7] = elmntp->nps[n_h[11]] ;
802                         }
803                 }
804                 if(((elmntp->nps[n_h[6]] == NULL)  || (elmntp->nps[n_h[5]] ==
805     NULL))
806                  && (elmntp->nps[n_h[9]] == NULL)){
807                         theElp->vtype = 2342 ;
808                         theElp->nnds = 10 ;
809                         theElp->nps[0] = elmntp->nps[n_h[0]] ;
810                         theElp->nps[1] = elmntp->nps[n_h[1]] ;
811                         theElp->nps[2] = elmntp->nps[n_h[2]] ;
812                         theElp->nps[3] = elmntp->nps[n_h[3]] ;
813                         theElp->nps[4] = elmntp->nps[n_h[11]] ;
814                         theElp->nps[5] = elmntp->nps[n_h[15]] ;
815                         theElp->nps[6] = elmntp->nps[n_h[14]] ;
816                         theElp->nps[7] = elmntp->nps[n_h[10]] ;
817                         theElp->nps[8] = elmntp->nps[n_h[7]] ;
818                         theElp->nps[9] = elmntp->nps[n_h[4]] ;
```

```
819                 }
820                 if((elmntp->nps[n_h[10]] == NULL) || (elmntp->nps[n_h[11]] ==
821   NULL)){
822                         theElp->vtype = 233 ;
823                         theElp->nnds = 8 ;
824                         theElp->nps[0] = elmntp->nps[n_h[0]] ;
825                         theElp->nps[1] = elmntp->nps[n_h[1]] ;
826                         theElp->nps[2] = elmntp->nps[n_h[2]] ;
827                         theElp->nps[3] = elmntp->nps[n_h[3]] ;
828                         theElp->nps[4] = elmntp->nps[n_h[4]] ;
829                         theElp->nps[5] = elmntp->nps[n_h[5]] ;
830                         theElp->nps[6] = elmntp->nps[n_h[6]] ;
831                         theElp->nps[7] = elmntp->nps[n_h[7]] ;
832
833                         if ((elmntp->nps[n_h[6]] == NULL)  || (elmntp->nps[n_h[5]]
834   == NULL)){
835                                 theElp->vtype = 230;
836                                 theElp->nnds = 6 ;
837                                 theElp->nps[0] = elmntp->nps[n_h[0]] ;
838                                 theElp->nps[1] = elmntp->nps[n_h[1]] ;
839                                 theElp->nps[2] = elmntp->nps[n_h[2]] ;
840                                 theElp->nps[3] = elmntp->nps[n_h[3]] ;
841                                 theElp->nps[4] = elmntp->nps[n_h[4]] ;
842                                 theElp->nps[5] = elmntp->nps[n_h[7]] ;
843                         }
844                 }
845
846                 if((elmntp->nps[n_h[4]] == NULL) || (elmntp->nps[n_h[7]] ==
847   NULL)){
848                         theElp->vtype = 233 ;
849                         theElp->nnds = 8 ;
850                         theElp->nps[0] = elmntp->nps[n_h[1]] ;
851                         theElp->nps[1] = elmntp->nps[n_h[2]] ;
852                         theElp->nps[2] = elmntp->nps[n_h[3]] ;
853                         theElp->nps[3] = elmntp->nps[n_h[0]] ;
854                         theElp->nps[4] = elmntp->nps[n_h[10]] ;
855                         theElp->nps[5] = elmntp->nps[n_h[14]] ;
856                         theElp->nps[6] = elmntp->nps[n_h[15]] ;
857                         theElp->nps[7] = elmntp->nps[n_h[11]] ;
858
859                         if ((elmntp->nps[n_h[14]] == NULL) || (elmntp-
860   >nps[n_h[15]] == NULL)){
861                                 theElp->vtype = 230;
862                                 theElp->nnds = 6 ;
863                                 theElp->nps[0] = elmntp->nps[n_h[1]] ;
864                                 theElp->nps[1] = elmntp->nps[n_h[2]] ;
865                                 theElp->nps[2] = elmntp->nps[n_h[3]] ;
866                                 theElp->nps[3] = elmntp->nps[n_h[0]] ;
867                                 theElp->nps[4] = elmntp->nps[n_h[10]] ;
868                                 theElp->nps[5] = elmntp->nps[n_h[11]] ;
869                         }
870                 }
871                 if(((elmntp->nps[n_h[10]] == NULL) || (elmntp->nps[n_h[11]] ==
872   NULL))
873                         && ((elmntp->nps[n_h[7]] == NULL)  || (elmntp->nps[n_h[4]]
874   == NULL))){
875
876                         theElp->vtype = 211;
877                         theElp->nnds = 4 ;
878                         theElp->nps[0] = elmntp->nps[n_h[0]] ;
879                         theElp->nps[1] = elmntp->nps[n_h[1]] ;
880                         theElp->nps[2] = elmntp->nps[n_h[2]] ;
881                         theElp->nps[3] = elmntp->nps[n_h[3]] ;
```

```
882                     }
883
884             return(-1) ;
885     }
```

```
1   #include "fe.h"
2
3
4   int                 get_gspts(i_type,g)
5   int                 i_type ;
6   struct gausspts             g[] ;
7
8   {
9           int n ;
10
11          switch (i_type) {
12
13                  case 1 :
14                          n = gs_pt(g) ;
15                          break ;
16
17                  case 111 :
18                          n = gs_1D2(g) ;
19                          break ;
20
21                  case 119 :
22                          n = qu_1D2(g) ;
23                          break ;
24
25                  case 118 :
26                          n = qu_1D2r(g) ;
27                          break ;
28
29                  case 128 :
30                          n = gs_1D2(g) ;
31                          break ;
32
33                  case 129 :
34                          n = gs_1D2(g) ;
35                          break ;
36
37                  case 121 :
38                          n = gs_1D3(g) ;
39                          break ;
40
41                  case 131 :
42                          n = gs_1D3(g) ;
43                          break ;
44
45                  case 138 :
46                          n = gs_1D3(g) ;
47                          break ;
48
49                  case 139 :
50                          n = gs_1D3(g) ;
51                          break ;
52
53                  case 149 :
54                          n = gs_1D3(g) ;
55                          break ;
56
57                  case 159 :
58                          n = gs_1D3(g) ;
59                          break ;
60
61                  case 169 :
62                          n = gs_1D5(g) ;
63                          break ;
```

```
64
65                  case 179 :
66                        n = gs_1D5(g) ;
67                        break ;
68
69                  case 175 :
70                        n = gs_1D5(g) ;
71                        break ;
72
73                  case 210 :
74                        n = gs_2t2(g) ;
75                        break ;
76
77                  case 220 :
78                        n = gs_2t2(g) ;
79                        break ;
80
81                  case 211 :
82                        n = gs_2D2(g) ;
83                        break ;
84
85                  case 216 :
86                        n = gs_2D2(g) ;
87                        break ;
88
89                  case 221 :
90                        n = gs_2D2(g) ;
91                        break ;
92
93                  case 226 :
94                        n = gs_2D2(g) ;
95                        break ;
96
97                  case 222 :
98                        n = gs_2D2(g) ;
99                        break ;
100
101                 case 227 :
102                       n = gs_2D2(g) ;
103                       break ;
104
105                 case 219 :
106                       n = qu_2D2(g) ;
107                       break ;
108
109                 case 228 :
110                       n = qu_2D3(g) ;
111                       break ;
112
113                 case 229 :
114                       n = qu_2D3(g) ;
115                       break ;
116
117                 case 231 :
118                       n = gs_2D3(g) ;
119                       break ;
120
121                 case 230 :
122                       n = gs_2D2(g) ;
123                       break ;
124
125                 case 2311 :
126                       n = gs_2D2(g) ;
```

```
127                     break ;
128
129             case 232 :
130                     n = gs_2D2(g) ;
131                     break ;
132
133             case 233 :
134                     n = gs_2D3(g) ;
135                     break ;
136
137             case 2341 :
138                     n = gs_2D3(g) ;
139                     break ;
140
141             case 2342 :
142                     n = gs_2D3(g) ;
143                     break ;
144
145             case 2351 :
146                     n = gs_2D3(g) ;
147                     break ;
148
149             case 2353 :
150                     n = gs_2D3(g) ;
151                     break ;
152
153             case 2352 :
154                     n = gs_2D3(g) ;
155                     break ;
156
157             case 2361 :
158                     n = gs_2D3(g) ;
159                     break ;
160
161             case 2363 :
162                     n = gs_2D3(g) ;
163                     break ;
164
165             case 2362 :
166                     n = gs_2D3(g) ;
167                     break ;
168
169             case 2371 :
170                     n = gs_2D3(g) ;
171                     break ;
172
173             case 2372 :
174                     n = gs_2D3(g) ;
175                     break ;
176
177             case 238 :
178                     n = gs_2D3(g) ;
179                     break ;
180
181             case 239 :
182                     n = gs_2D3(g) ;
183                     break ;
184
185             case 235 :
186                     n = gs_2D2(g) ;
187                     break ;
188
189             case 911 :
```

```
190                         n = gs_1D5(g) ;
191                         break ;
192
193                 case 918 :
194                         n = gs_qu1D5r(g) ;
195                         break ;
196
197                 case 919 :
198                         n = gs_qu1D5(g) ;
199                         break ;
200
201                 case 921 :
202                         n = gs_1D5(g) ;
203                         break ;
204
205                 case 929 :
206                         n = gs_qu1D5(g) ;
207                         break ;
208
209                 case 922 :
210                         n = gs_2D4(g) ;
211                         break ;
212
213                 default :
214                         n = 0 ;
215                         break ;
216         }
217         return(n) ;
218 }
219
220 int                     gs_pt(g)
221 struct gausspts                 g[] ;
222
223 {
224         g[0].x =  0.0 ;
225         g[0].y =  0.0 ;
226         g[0].z =  0.0 ;
227         g[0].w =  1.0 ;
228
229         return(1) ;
230 }
231
232 int                     gs_1D1(g)
233 struct gausspts                 g[] ;
234
235 {
236         g[0].x =  0.0 ;
237         g[0].y =  0.0 ;
238         g[0].z =  0.0 ;
239         g[0].w =  2.0 ;
240
241         return(1) ;
242 }
243
244 int                     gs_1D2(g)
245 struct gausspts                 g[] ;
246
247 {
248         g[0].x = -0.577350269189626 ;
249         g[0].y =  0.0 ;
250         g[0].z =  0.0 ;
251         g[0].w =  1.0 ;
252
```

```
253         g[1].x =  0.577350269189626 ;
254         g[1].y =  0.0 ;
255         g[1].z =  0.0 ;
256         g[1].w =  1.0 ;
257
258         return(2) ;
259     }
260
261     int                 qu_1D2(g)
262     struct gausspts             g[] ;
263
264     {
265         g[0].x =  0.788675134594813 ;
266         g[0].y =  0.0 ;
267         g[0].z =  0.0 ;
268         g[0].w =  0.5 ;
269
270         g[1].x =  0.211324865405187 ;
271         g[1].y =  0.0 ;
272         g[1].z =  0.0 ;
273         g[1].w =  0.5 ;
274
275         return(2) ;
276     }
277
278     int                 qu_1D2r(g)
279     struct gausspts             g[] ;
280
281     {
282         g[0].x =  -0.788675134594813 ;
283         g[0].y =  0.0 ;
284         g[0].z =  0.0 ;
285         g[0].w =  0.5 ;
286
287         g[1].x =  -0.211324865405187 ;
288         g[1].y =  0.0 ;
289         g[1].z =  0.0 ;
290         g[1].w =  0.5 ;
291
292         return(2) ;
293     }
294
295     int                 gs_1D3(g)
296     struct gausspts             g[] ;
297
298     {
299         g[0].x = -0.774596669241483 ;
300         g[0].y =  0.0 ;
301         g[0].z =  0.0 ;
302         g[0].w =  0.555555555555556 ;
303
304         g[1].x =  0.0 ;
305         g[1].y =  0.0 ;
306         g[1].z =  0.0 ;
307         g[1].w =  0.888888888888889 ;
308
309         g[2].x =  0.774596669241483 ;
310         g[2].y =  0.0 ;
311         g[2].z =  0.0 ;
312         g[2].w =  0.555555555555556 ;
313
314         return(3) ;
315     }
```

```
316
317   int                     gs_1D5(g)
318   struct gausspts                 g[] ;
319
320   {
321           g[0].x = -0.906179845938664 ;
322           g[0].y =  0.0 ;
323           g[0].z =  0.0 ;
324           g[0].w =  0.236926885056189 ;
325
326           g[1].x = -0.538469310105683 ;
327           g[1].y =  0.0 ;
328           g[1].z =  0.0 ;
329           g[1].w =  0.478628670499366;
330
331           g[2].x =  0.0 ;
332           g[2].y =  0.0 ;
333           g[2].z =  0.0 ;
334           g[2].w =  0.568888888888889 ;
335
336           g[3].x =  0.538469310105683 ;
337           g[3].y =  0.0 ;
338           g[3].z =  0.0 ;
339           g[3].w =  0.478628670499366 ;
340
341           g[4].x =  0.906179845938664 ;
342           g[4].y =  0.0 ;
343           g[4].z =  0.0 ;
344           g[4].w =  0.236926885056189 ;
345
346           return(5) ;
347   }
348
349   int                     gs_qu1D5(g)
350   struct gausspts                 g[] ;
351
352   {
353           g[0].x =  0.046910077030668 ;
354           g[0].y =  0.0 ;
355           g[0].z =  0.0 ;
356           g[0].w =  0.118463442528945 ;
357
358           g[1].x =  0.2307653449471585 ;
359           g[1].y =  0.0 ;
360           g[1].z =  0.0 ;
361           g[1].w =  0.239314335249683;
362
363           g[2].x =  0.5 ;
364           g[2].y =  0.0 ;
365           g[2].z =  0.0 ;
366           g[2].w =  0.284444444444445 ;
367
368           g[3].x =  0.7692346550528415 ;
369           g[3].y =  0.0 ;
370           g[3].z =  0.0 ;
371           g[3].w =  0.239314335249683 ;
372
373           g[4].x =  0.953089922969332 ;
374           g[4].y =  0.0 ;
375           g[4].z =  0.0 ;
376           g[4].w =  0.118463442528945 ;
377
378           return(5) ;
```

```
379  }
380
381  int                    gs_qu1D5r(g)
382  struct gausspts                g[] ;
383
384  {
385          g[0].x = -0.0469100770306680 ;
386          g[0].y =  0.0 ;
387          g[0].z =  0.0 ;
388          g[0].w =  0.1184634425289450 ;
389
390          g[1].x = -0.2307653449471585 ;
391          g[1].y =  0.0 ;
392          g[1].z =  0.0 ;
393          g[1].w =  0.2393143352496830;
394
395          g[2].x = -0.5 ;
396          g[2].y =  0.0 ;
397          g[2].z =  0.0 ;
398          g[2].w =  0.2844444444444445 ;
399
400          g[3].x = -0.7692346550528415 ;
401          g[3].y =  0.0 ;
402          g[3].z =  0.0 ;
403          g[3].w =  0.2393143352496830 ;
404
405          g[4].x = -0.9530899229693320 ;
406          g[4].y =  0.0 ;
407          g[4].z =  0.0 ;
408          g[4].w =  0.1184634425289450 ;
409
410          return(5) ;
411  }
412
413  int                    gs_2t1(g)
414  struct gausspts                g[] ;
415
416  {
417          g[0].x =  0.3333333333333 ;
418          g[0].y =  0.3333333333333 ;
419          g[0].z =  0.0 ;
420          g[0].w =  0.5 ;
421
422          return(1) ;
423  }
424
425  int                    gs_2D1(g)
426  struct gausspts                g[] ;
427
428  {
429          g[0].x =  0.0 ;
430          g[0].y =  0.0 ;
431          g[0].z =  0.0 ;
432          g[0].w =  4.0 ;
433
434          return(1) ;
435  }
436
437  int                    gs_2t2(g)
438  struct gausspts                g[] ;
439
440  {
441          g[0].x =  0.5 ;
```

```
442         g[0].y =  0.5 ;
443         g[0].z =  0.0 ;
444         g[0].w =  0.166666666666667 ;
445
446         g[1].x =  0.0 ;
447         g[1].y =  0.5 ;
448         g[1].z =  0.0 ;
449         g[1].w =  0.166666666666667 ;
450
451         g[2].x =  0.5 ;
452         g[2].y =  0.0 ;
453         g[2].z =  0.0 ;
454         g[2].w =  0.166666666666667 ;
455
456         return(3) ;
457   }
458
459   int                   gs_2D2(g)
460   struct gausspts             g[] ;
461
462   {
463         g[0].x =  0.577350269189626 ;
464         g[0].y = -0.577350269189626 ;
465         g[0].z =  0.0 ;
466         g[0].w =  1.0 ;
467
468         g[1].x =  0.577350269189626 ;
469         g[1].y =  0.577350269189626 ;
470         g[1].z =  0.0 ;
471         g[1].w =  1.0 ;
472
473         g[2].x = -0.577350269189626 ;
474         g[2].y =  0.577350269189626 ;
475         g[2].z =  0.0 ;
476         g[2].w =  1.0 ;
477
478         g[3].x = -0.577350269189626 ;
479         g[3].y = -0.577350269189626 ;
480         g[3].z =  0.0 ;
481         g[3].w =  1.0 ;
482
483         return(4) ;
484   }
485
486
487   int                   qu_2D2(g)
488   struct gausspts             g[] ;
489
490   {
491         g[0].x =  0.7886751345 ;
492         g[0].y =  0.2113248655 ;
493         g[0].z =  0.0 ;
494         g[0].w =  0.25 ;
495
496         g[1].x =  0.7886751345 ;
497         g[1].y =  0.7886751345 ;
498         g[1].z =  0.0 ;
499         g[1].w =  0.25 ;
500
501         g[2].x =  0.2113248655 ;
502         g[2].y =  0.7886751345 ;
503         g[2].z =  0.0 ;
504         g[2].w =  0.25 ;
```

```
505
506         g[3].x =  0.2113248655 ;
507         g[3].y =  0.2113248655 ;
508         g[3].z =  0.0 ;
509         g[3].w =  0.25 ;
510
511         return(4) ;
512   }
513
514   int                   gs_2D3b2(g)
515   struct gausspts             g[] ;
516
517   {
518         g[0].x =  0.774596669241483 ;
519         g[0].y = -0.577350269189626 ;
520         g[0].z =  0.0 ;
521         g[0].w =  0.555555555555556 ;
522
523         g[1].x =  0.774596669241483 ;
524         g[1].y =  0.577350269189626 ;
525         g[1].z =  0.0 ; ——
526         g[1].w =  0.555555555555556 ;
527
528         g[2].x =  0.0 ;
529         g[2].y = -0.577350269189626 ;
530         g[2].z =  0.0 ;
531         g[2].w =  0.888888888888889 ;
532
533         g[3].x =  0.0 ;
534         g[3].y =  0.577350269189626 ;
535         g[3].z =  0.0 ;
536         g[3].w =  0.888888888888889 ;
537
538         g[4].x = -0.774596669241483 ;
539         g[4].y = -0.577350269189626 ;
540         g[4].z =  0.0 ;
541         g[4].w =  0.555555555555556 ;
542
543         g[5].x = -0.774596669241483 ;
544         g[5].y =  0.577350269189626 ;
545         g[5].z =  0.0 ;
546         g[5].w =  0.555555555555556 ;
547
548         return(6) ;
549   }
550
551   int                   gs_2D3(g)
552   struct gausspts             g[] ;
553
554   {/
555         g[0].x =  0.774596669241483 ;
556         g[0].y = -0.774596669241483 ;
557         g[0].z =  0.0 ;
558         g[0].w =  0.3086419 ;
559
560         g[1].x =  0.774596669241483 ;
561         g[1].y =  0.0 ;
562         g[1].z =  0.0 ;
563         g[1].w =  0.4938271 ;
564
565         g[2].x =  0.774596669241483 ;
566         g[2].y =  0.774596669241483 ;
567         g[2].z =  0.0 ;
```

```
568         g[2].w =  0.3086419 ;
569
570         g[3].x =  0.0 ;
571         g[3].y =  0.774596669241483 ;
572         g[3].z =  0.0 ;
573         g[3].w =  0.4938271 ;
574
575         g[4].x =  0.0 ;
576         g[4].y =  0.0 ;
577         g[4].z =  0.0 ;
578         g[4].w =  0.7901233 ;
579
580         g[5].x =  0.0 ;
581         g[5].y = -0.774596669241483 ;
582         g[5].z =  0.0 ;
583         g[5].w =  0.4938271 ;
584
585         g[6].x = -0.774596669241483 ;
586         g[6].y = -0.774596669241483 ;
587         g[6].z =  0.0 ;
588         g[6].w =  0.3086419 ;
589
590         g[7].x = -0.774596669241483 ;
591         g[7].y =  0.0 ;
592         g[7].z =  0.0 ;
593         g[7].w =  0.4938271 ;
594
595         g[8].x = -0.774596669241483 ;
596         g[8].y =  0.774596669241483 ;
597         g[8].z =  0.0 ;
598         g[8].w =  0.3086419 ;
599
600         return(9) ;
601   }
602   int                   qu_2D3(g)
603   struct gausspts             g[] ;
604
605   {
606         g[0].x =  0.112701665379259 ;
607         g[0].y =  0.112701665379259 ;
608         g[0].z =  0.0 ;
609         g[0].w =  0.077160475 ;
610
611         g[1].x =  0.112701665379259 ;
612         g[1].y =  0.5 ;
613         g[1].z =  0.0 ;
614         g[1].w =  0.12345678 ;
615
616         g[2].x =  0.112701665379259 ;
617         g[2].y =  0.887298334621741 ;
618         g[2].z =  0.0 ;
619         g[2].w =  0.077160475 ;
620
621         g[3].x =  0.5 ;
622         g[3].y =  0.112701665379259 ;
623         g[3].z =  0.0 ;
624         g[3].w =  0.12345678 ;
625
626         g[4].x =  0.5 ;
627         g[4].y =  0.5 ;
628         g[4].z =  0.0 ;
629         g[4].w =  0.197530825 ;
630
```

```
631         g[5].x =  0.5 ;
632         g[5].y =  0.887298334621741 ;
633         g[5].z =  0.0 ;
634         g[5].w =  0.12345678 ;
635
636         g[6].x =  0.887298334621741 ;
637         g[6].y =  0.112701665379259 ;
638         g[6].z =  0.0 ;
639         g[6].w =  0.077160475 ;
640
641         g[7].x =  0.887298334621741 ;
642         g[7].y =  0.5 ;
643         g[7].z =  0.0 ;
644         g[7].w =  0.12345678 ;
645
646         g[8].x =  0.887298334621741 ;
647         g[8].y =  0.887298334621741 ;
648         g[8].z =  0.0 ;
649         g[8].w =  0.077160475 ;
650
651         return(9) ;
652   }
653
654   int                gs_2D4(g)
655   struct gausspts            g[] ;
656
657   {
658         g[0].x =  0.861136311594053 ;
659         g[0].y = -0.861136311594053 ;
660         g[0].z =  0.0 ;
661         g[0].w =  0.1210029932 ;
662
663         g[1].x =  0.861136311594053 ;
664         g[1].y =  -0.339981043584856 ;
665         g[1].z =  0.0 ;
666         g[1].w =  0.2268518518 ;
667
668         g[2].x =  0.861136311594053 ;
669         g[2].y =  0.339981043584856 ;
670         g[2].z =  0.0 ;
671         g[2].w =  0.2268518518 ;
672
673         g[3].x =  0.861136311594053 ;
674         g[3].y =  0.861136311594053 ;
675         g[3].z =  0.0 ;
676         g[3].w =  0.1210029932 ;
677
678         g[4].x =  0.339981043584856 ;
679         g[4].y =  0.861136311594053 ;
680         g[4].z =  0.0 ;
681         g[4].w =  0.2268518518 ;
682
683         g[5].x =  -0.339981043584856 ;
684         g[5].y =  0.861136311594053 ;
685         g[5].z =  0.0 ;
686         g[5].w =  0.2268518518 ;
687
688         g[6].x = -0.861136311594053 ;
689         g[6].y = 0.861136311594053 ;
690         g[6].z =  0.0 ;
691         g[6].w =  0.1210029932 ;
692
693         g[7].x = -0.861136311594053 ;
```

```
694         g[7].y =  0.339981043584856 ;
695         g[7].z =  0.0 ;
696         g[7].w =  0.2268518518 ;
697
698         g[8].x = -0.861136311594053 ;
699         g[8].y = -0.339981043584856 ;
700         g[8].z =  0.0 ;
701         g[8].w =  0.2268518518 ;
702
703         g[9].x = -0.861136311594053 ;
704         g[9].y = -0.861136311594053 ;
705         g[9].z =  0.0 ;
706         g[9].w =  0.1210029932 ;
707
708         g[10].x = -0.339981043584856 ;
709         g[10].y = -0.861136311594053 ;
710         g[10].z =  0.0 ;
711         g[10].w =  0.2268518518 ;
712
713         g[11].x = 0.339981043584856 ;
714         g[11].y = -0.861136311594053 ;
715         g[11].z =  0.0 ;
716         g[11].w =  0.2268518518 ;
717
718         g[12].x = 0.339981043584856 ;
719         g[12].y = -0.339981043584856 ;
720         g[12].z =  0.0 ;
721         g[12].w =  0.4252933019 ;
722
723         g[13].x = 0.339981043584856 ;
724         g[13].y = 0.339981043584856 ;
725         g[13].z =  0.0 ;
726         g[13].w =  0.4252933019 ;
727
728         g[14].x = -0.339981043584856 ;
729         g[14].y = 0.339981043584856 ;
730         g[14].z =  0.0 ;
731         g[14].w =  0.4252933019 ;
732
733         g[15].x = -0.339981043584856 ;
734         g[15].y = -0.339981043584856 ;
735         g[15].z =  0.0 ;
736         g[15].w =  0.4252933019 ;
737
738
739         return(16) ;
740     }
```

```
#include "fe.h"


extern struct control  N ;


int                     get_shape(el_type,x,psf)
int                     el_type ;
struct gausspts                 *x ;
struct shapefuncs       *psf ;

{
        int     err ;
        switch (el_type) {

                case 210 :
                        err = shap_2t1(x,psf) ;
                        break ;

                case 211 :
                        err = shap_2s1(x,psf) ;
                        break ;

                case 216 :
                        err = shap_2s1(x,psf) ;
                        break ;

                case 220 :
                        err = shap_2t2(x,psf) ;
                        break ;

                case 221 :
                        err = shap_2s2(x,psf) ;
                        break ;

                case 226 :
                        err = shap_2s2(x,psf) ;
                        break ;

                case 222 :
                        err = shap_212(x,psf) ;
                        break ;

                case 227 :
                        err = shap_212(x,psf) ;
                        break ;

                case 228 :
                        err = shqu6_212(x,psf) ;
                        break ;

                case 229 :
                        err = shqu9_212(x,psf) ;
                        break ;

                case 224 :
                        err = shqu8_212(x,psf) ;
                        break ;

                case 219 :
                        err = shqu_211(x,psf) ;
                        break ;

```

```
64                  case 231 :
65                        err = shap_2s3(x,psf) ;
66                        break ;
67
68                  case 111 :
69                        err = shap_1s1(x,psf) ;
70                        break ;
71
72                  case 121 :
73                        err = shap_1s2(x,psf) ;
74                        break ;
75
76                  case 128 :
77                        err = shcu3r_1s2(x,psf) ;
78                        break ;
79
80                  case 129 :
81                        err = shcu3_1s2(x,psf) ;
82                        break ;
83
84                  case 139 :
85                        err = shapcu_1s3(x,psf) ;
86                        break ;
87
88                  case 138 :
89                        err = shapcur_1s3(x,psf) ;
90                        break ;
91
92                  case 131 :
93                        err = shap_1s3(x,psf) ;
94                        break ;
95
96                  case 1 :
97                        err = shap_pt(x,psf) ;
98                        break ;
99
100                 case 230 :
101                       err = shcu2_2x1(x,psf) ;
102                       break ;
103
104                 case 2311 :
105                       err = shcu3_2x2(x,psf) ;
106                       break ;
107
108                 case 232 :
109                       err = shcu4_2x2(x,psf) ;
110                       break ;
111
112                 case 233 :
113                       err = shcu3_3x1(x,psf) ;
114                       break ;
115
116                 case 2341 :
117                       err = shcu4_3x2(x,psf) ;
118                       break ;
119
120                 case 2342 :
121                       err = sh4_3x2b(x,psf) ;
122                       break ;
123
124                 case 2351 :
125                       err = shcu5_3x2(x,psf) ;
126                       break ;
```

```
127
128                 case 2353 :
129                         err = sh5_3x2b(x,psf) ;
130                         break ;
131
132                 case 2352 :
133                         err = shcu5_3x3(x,psf) ;
134                         break ;
135
136                 case 2361 :
137                         err = shcu6_3x2(x,psf) ;
138                         break ;
139
140                 case 2363 :
141                         err = sh6_3x2b(x,psf) ;
142                         break ;
143
144                 case 2362 :
145                         err = shcu6_3x3(x,psf) ;
146                         break ;
147
148                 case 2371 :
149                         err = shcu7_3x3(x,psf) ;
150                         break ;
151
152                 case 2372 :
153                         err = sh7_3x3b(x,psf) ;
154                         break ;
155
156                 case 238 :
157                         err = shcu8_3x3(x,psf) ;
158                         break ;
159
160                 case 239 :
161                         err = shcu9_3x3(x,psf) ;
162                         break ;
163
164                 case 235 :
165                         err = hermite(x,psf) ;
166                         break ;
167         }
168         return(err) ;
169 }
170
171
172 int     nsf(el_type)
173 int     el_type ;
174
175 {
176         struct gausspts             x ;
177         struct shapefuncs    psf ;
178         int                  err ;
179
180         x.x = 100.0 ;
181         x.y = 100.0 ;
182         x.z = 100.0 ;
183         err = get_shape(el_type,&x,&psf) ;
184         return(err) ;
185 }
186
187
188 int     nbounds(el_type)
189 int     el_type ;
```

```
190
191     {
192             int     err ;
193
194             err = 0 ;
195             switch (el_type) {
196
197                     case 210 :
198                             err = 3 ;
199                             break ;
200
201                     case 211 :
202                             err = 4 ;
203                             break ;
204
205                     case 216 :
206                             err = 4 ;
207                             break ;
208
209                     case 220 :
210                             err = 6 ;
211                             break ;
212
213                     case 221 :
214                             err = 8 ;
215                             break ;
216
217                     case 222 :
218                             err = 8 ;
219                             break ;
220
221                     case 226 :
222                             err = 8 ;
223                             break ;
224
225                     case 227 :
226                             err = 8 ;
227                             break ;
228
229                     case 224 :
230                             err = 4 ;
231                             break ;
232
233                     case 228 :
234                             err = 4 ;
235                             break ;
236
237                     case 229 :
238                             err = 4 ;
239                             break ;
240
241                     case 239 :
242                             err = 4 ;
243                             break ;
244
245                     case 230 :
246                             err = 4 ;
247                             break ;
248
249                     case 2311 :
250                             err = 4 ;
251                             break ;
252
```

```
253                 case 232 :
254                       err = 4 ;
255                       break ;
256
257                 case 233 :
258                       err = 4 ;
259                       break ;
260
261                 case 2341 :
262                       err = 4 ;
263                       break ;
264
265                 case 2342 :
266                       err = 4 ;
267                       break ;
268
269                 case 2351 :
270                       err = 4 ;
271                       break ;
272
273                 case 2353 :
274                       err = 4 ;
275                       break ;
276
277                 case 2352 :
278                       err = 4 ;
279                       break ;
280
281                 case 2361 :
282                       err = 4 ;
283                       break ;
284
285                 case 2363 :
286                       err = 4 ;
287                       break ;
288
289                 case 2362 :
290                       err = 4 ;
291                       break ;
292
293                 case 2371 :
294                       err = 4 ;
295                       break ;
296
297                 case 2372 :
298                       err = 4 ;
299                       break ;
300
301                 case 238 :
302                       err = 4 ;
303                       break ;
304
305                 case 231 :
306                       err = 12 ;
307                       break ;
308
309                 case 111 :
310                       err = 2 ;
311                       break ;
312
313                 case 118 :
314                       err = 2 ;
315                       break ;
```

```
316
317                 case 119 :
318                       err = 2 ;
319                       break ;
320
321                 case 128 :
322                       err = 2 ;
323                       break ;
324
325                 case 129 :
326                       err = 2 ;
327                       break ;
328
329                 case 139 :
330                       err = 2 ;
331                       break ;
332
333                 case 138 :
334                       err = 2 ;
335                       break ;
336
337                 case 121 :
338                       err = 2 ;
339                       break ;
340
341                 case 131 :
342                       err = 2 ;
343                       break ;
344
345                 case 1 :
346                       err = 1 ;
347                       break ;
348
349                 case 235 :
350                       err = 4 ;
351                       break ;
352           }
353           return(err) ;
354     }
355
356
```

```
1   #include "fe.h"
2
3
4   extern struct control  N ;
5   int     shap_2t1(x,p)
6   struct gausspts             *x ;
7   struct shapefuncs    *p ;
8
9   {
10          double r,s,mr,ms,pr,ps,t ;
11
12          r = x->x ;
13          s = x->y ;
14          t = 1.0 - r - s ;
15          if( (r<-0.0001) || (r>1.0001) || (s<-0.00001) || (s>1.00001) || (t<-
16  0.00001) ) return(3) ;
17
18          p->dof = 3 ;
19
20          p->f[0] = r ;
21          p->f[1] = s ;
22          p->f[2] = t ;
23
24          if(x->w < 0.0)
25                return(0) ;
26
27          p->dfdr[0] =  1.0 ;
28          p->dfdr[1] =  0.0 ;
29          p->dfdr[2] = -1.0 ;
30
31          p->dfds[0] =  0.0 ;
32          p->dfds[1] =  1.0 ;
33          p->dfds[2] = -1.0 ;
34
35          return(0) ;
36  }
37
38
39  int     shap_2s1(x,p)
40  struct gausspts             *x ;
41  struct shapefuncs    *p ;
42
43  {
44          double r,s,mr,ms,pr,ps,t ;
45
46          r = x->x ;
47          s = x->y ;
48          if( (r<-1.00001) || (r>1.00001) || (s<-1.00001) || (s>1.00001) )
49  return(4) ;
50
51          t = 0.25 ;
52          mr = t * (1.0 - r) ;
53          ms = 1.0 - s ;
54          pr = t * (1.0 + r) ;
55          ps = 1.0 + s ;
56
57          p->dof = 4 ;
58
59          p->f[0] = pr * ms ;
60          p->f[1] = pr * ps ;
61          p->f[2] = mr * ps ;
62          p->f[3] = mr * ms ;
63
```

```
64          if(x->w < 0.0)
65                  return(0) ;
66          ms *= t ;
67          ps *= t ;
68
69          p->dfdr[0] =  ms ;
70          p->dfdr[1] =  ps ;
71          p->dfdr[2] = -ps ;
72          p->dfdr[3] = -ms ;
73
74          p->dfds[0] = -pr ;
75          p->dfds[1] =  pr ;
76          p->dfds[2] =  mr ;
77          p->dfds[3] = -mr ;
78
79          return(0) ;
80  }
81
82  int     shcu3_ls2(x,p)
83  struct gausspts             *x ;
84  struct shapefuncs     *p ;
85
86  {
87          double r, mr, pr, rp3, t, tt ;
88
89          r = x->x ;
90          if( (r<-1.0) || (r>1.0) ) return(3) ;
91
92          t = 0.125 ;
93          tt = 0.25 ;
94          mr = 1.0 - r ;
95          pr = 1.0 + r ;
96          rp3 = 3.0 + r ;
97
98          p->dof = 3 ;
99
100         p->f[0] = tt * rp3 * mr ;
101         p->f[1] = t * rp3 * pr ;
102         p->f[2] = -t * pr * mr ;
103
104         if(x->w < 0.0)
105                 return(0) ;
106
107         p->dfdr[0] = -0.5 * pr ;
108         p->dfdr[1] = tt *(2.0+r)  ;
109         p->dfdr[2] = tt * r ;
110
111 /*      p->d2fdr[0] = -0.5 ;
112         p->d2fdr[1] = tt  ;
113         p->d2fdr[2] = tt ;
114 */
115         return(0) ;
116 }
117 int     shapcu_ls3(x,p)
118 struct gausspts             *x ;
119 struct shapefuncs     *p ;
120
121 {
122         double r,rm,rp,rp3,rp5,t ;
123
124         r = x->x ;
125         if( (r<-1.0) || (r>1.0) ) return(4) ;
126
```

```
127         t = 1.0/48.0  ;
128         rm = 1.0 - r ;
129         rp = 1.0 + r ;
130         rp3 = r + 3.0 ;
131         rp5 = r + 5.0 ;
132
133         p->dof = 4 ;
134
135         p->f[0] =  3.0*t * rm * rp3 * rp5 ;
136         p->f[1] =  t * rp * rp3 * rp5 ;
137         p->f[2] = -3.0*t * rp * rm * rp5 ;
138         p->f[3] = t * rp * rm * rp3 ;
139
140         if(x->w < 0.0)
141               return(0) ;
142
143         p->dfdr[0] = -3.0 * t * ( 7.0 + (14.0 + 3.0*r)*r) ;
144         p->dfdr[1] = t * ( 23.0 + r * (18.0 + 3.0*r))  ;
145         p->dfdr[2] = -3.0 * t * ( 1.0 - r*(10.0 +3.0*r)) ;
146         p->dfdr[3] = t * ( 1.0 -3.0*r*(2.0 + r)) ;
147
148         return(0) ;
149   }
150   int     shcu3r_1s2(x,p)
151   struct gausspts             *x ;
152   struct shapefuncs    *p ;
153
154   {
155         double r,mr,pr,t ;
156
157         r = x->x ;
158         if( (r<-1.0) || (r>1.0) ) return(3) ;
159
160         t = 0.125 ;
161         mr = 1.0 - r ;
162         pr = 1.0 + r ;
163
164         p->dof = 3 ;
165
166         p->f[0] = t *(3.0- r) * mr ;
167         p->f[1] = 2*t* (3.0- r) * pr ;
168         p->f[2] = -t * pr * mr ;
169
170         if(x->w < 0.0)
171               return(0) ;
172
173         p->dfdr[0] = -2.0*t * (2.0-r) ;
174         p->dfdr[1] = 4.0 * t*mr  ;
175         p->dfdr[2] = 2.0*t * r ;
176
177         return(0) ;
178   )int    shapcur_1s3(x,p)
179   struct gausspts             *x ;
180   struct shapefuncs    *p ;
181
182   {
183         double r,rm,rp,rp3,rp5,t ;
184
185         r = x->x ;
186         if( (r<-1.0) || (r>1.0) ) return(4) ;
187
188         t = 1.0/48.0  ;
189         rm = 1.0 - r ;
```

```
190         rp = 1.0 + r ;
191         rp3 = 3.0 - r ;
192         rp5 = 5.0 - r;
193
194         p->dof = 4 ;
195
196         p->f[0] = t * rm * rp3 * rp5 ;
197         p->f[1] =  3.0 * t * rp * rp3 * rp5 ;
198         p->f[2] = -3.0*t * rp * rm * rp5 ;
199         p->f[3] = t * rp * rm * rp3 ;
200
201         if(x->w < 0.0)
202               return(0) ;
203
204         p->dfdr[0] = t * ( -23.0 + (6.0 - r)*r*3.0) ;
205         p->dfdr[1] = 3.0 * t * (7.0 - r * (14.0 - 3.0*r))  ;
206         p->dfdr[2] = 3.0 * t * ( 1.0 + r*(10.0 -3.0*r)) ;
207         p->dfdr[3] = t * ( -1.0 -3.0*r*(2.0 - r)) ;
208
209         return(0) ;
210   }
211
212
213   int     shap_2t2(x,p)
214   struct gausspts             *x ;
215   struct shapefuncs     *p ;
216
217   {
218         double r,s,mr,ms,pr,ps,t ;
219
220         r = x->x ;
221         s = x->y ;
222         t = 1.0 - r - s ;
223         if( (r<-0.00001) || (r>1.00001) || (s<-0.00001) || (s>1.00001) || (t<-
224   0.00001) ) return(6) ;
225
226         p->dof = 6 ;
227
228         p->f[0] = r * (2.0*r - 1.0) ;
229         p->f[1] = 4.0 * r * s ;
230         p->f[2] = s * (2.0*s - 1.0) ;
231         p->f[3] = 4.0 * s * t ;
232         p->f[4] = t * (2.0*t - 1.0) ;
233         p->f[5] = 4.0 * r * t ;
234
235         if(x->w < 0.0)
236               return(0) ;
237
238         p->dfdr[0] =  4.0 * r - 1.0 ;
239         p->dfdr[1] =  4.0 * s ;
240         p->dfdr[2] =  0.0 ;
241         p->dfdr[3] = -4.0 * s ;
242         p->dfdr[4] = -4.0 * t + 1.0 ;
243         p->dfdr[5] =  4.0 * (t - r) ;
244
245         p->dfds[0] =  0.0 ;
246         p->dfds[1] =  4.0 * r ;
247         p->dfds[2] =  4.0 * s - 1.0 ;
248         p->dfds[3] =  4.0 * (t - s) ;
249         p->dfds[4] = -4.0 * t + 1.0 ;
250         p->dfds[5] = -4.0 * r ;
251
252         return(0) ;
```

```
253   }
254
255
256   int     shap_2s2(x,p)
257   struct gausspts             *x ;
258   struct shapefuncs     *p ;
259
260   {
261         double r,s,mr,ms,pr,ps,t,tr,ts,tprps,tmrps,tprms,tmrms ;
262         double trms,trps,tsmr,tspr,tps,tms,tpr,tmr ;
263
264         r = x->x ;
265         s = x->y ;
266         if( (r<-1.00001) || (r>1.00001) || (s<-1.00001) || (s>1.00001) )
267               return(8) ;
268
269         t = 0.25 ;
270         mr = 1.0 - r ;
271         ms = 1.0 - s ;
272         pr = 1.0 + r ;
273         ps = 1.0 + s ;
274         tprms = t * pr * ms ;
275         tprps = t * pr * ps ;
276         tmrps = t * mr * ps ;
277         tmrms = t * mr * ms ;
278
279         p->dof = 8 ;
280
281         p->f[0] = tprms * (r - ps) ;
282         p->f[2] = tprps * (r - ms) ;
283         p->f[4] = tmrps * (s - pr) ;
284         p->f[6] = tmrms * (-s - pr) ;
285         tprms *= 2.0 ;
286         tmrps *= 2.0 ;
287         p->f[1] = tprms * ps ;
288         p->f[3] = tmrps * pr ;
289         p->f[5] = tmrps * ms ;
290         p->f[7] = tprms * mr ;
291
292         if(x->w < 0.0)
293               return(0) ;
294
295         ts = 2.0 * s ;
296         tr = 2.0 * r ;
297         trms = tr - s ;
298         trps = tr + s ;
299         tsmr = ts - r ;
300         tspr = ts + r ;
301         tps = t * ps ;
302         tms = t * ms ;
303         tpr = t * pr ;
304         tmr = t * mr ;
305
306         p->dfdr[0] = tms * (trms) ;
307         p->dfdr[2] = tps * (trps) ;
308         p->dfdr[4] = tps * (trms) ;
309         p->dfdr[6] = tms * (trps) ;
310         p->dfdr[1] =  0.5 * ms * ps ;
311         p->dfdr[3] = -r * ps ;
312         p->dfdr[5] = -(p->dfdr[1]) ;
313         p->dfdr[7] = -r * ms ;
314
315         p->dfds[0] = tpr * (tsmr) ;
```

```
316         p->dfds[2] = tpr * (tspr) ;
317         p->dfds[4] = tmr * (tsmr) ;
318         p->dfds[6] = tmr * (tspr) ;
319         p->dfds[1] = -s * pr ;
320         p->dfds[3] = 0.5 * pr * mr ;
321         p->dfds[5] = -s * mr ;
322         p->dfds[7] = - (p->dfds[3]) ;
323
324         return(0) ;
325   }
326
327   int     shap_212(x,p)
328   struct gausspts             *x ;
329   struct shapefuncs    *p ;
330
331   {
332         double r,s,mr,ms,pr,ps,t,tr,ts,pmr,pms ;
333
334         r = x->x ;
335         s = x->y ;
336         if( (r<-1.00001) || (r>1.00001) || (s<-1.00001) || (s>1.00001) )
337               return(9) ;
338
339         t = 0.25 ;
340         tr = 2.0 * r ;
341         ts = 2.0 * s ;
342         mr = 1.0 - r ;
343         ms = 1.0 - s ;
344         pr = 1.0 + r ;
345         ps = 1.0 + s ;
346         pmr = pr * mr ;
347         pms = ps * ms ;
348
349         p->dof = 9 ;
350
351         p->f[0] = -t * pr * ms * r * s ;
352         p->f[2] = t * pr * ps * r * s ;
353         p->f[4] = -t * r * mr * s * ps ;
354         p->f[6] = t * r * mr * s * ms ;
355         p->f[1] = 0.5 * r * pr * pms ;
356         p->f[3] = 0.5 * pmr * s * ps ;
357         p->f[5] = -0.5 * r * mr * pms ;
358         p->f[7] = -0.5 * pmr * s * ms ;
359         p->f[8] = pmr * pms ;
360
361         if(x->w < 0.0)
362               return(0) ;
363
364         p->dfdr[0] = -t * s * ms * (tr + 1.0) ;
365         p->dfdr[2] = t * s * ps * (1.0 + tr) ;
366         p->dfdr[4] = -t * s * ps * (1.0 - tr) ;
367         p->dfdr[6] = t * s * ms * (1.0 - tr) ;
368         p->dfdr[1] =  0.5 * pms * (1.0 + tr) ;
369         p->dfdr[3] = -r * s * ps ;
370         p->dfdr[5] = -0.5 * (1.0 - tr) * pms ;
371         p->dfdr[7] =  r * s * ms ;
372         p->dfdr[8] = -2.0 * r * pms ;
373
374         p->dfds[0] = -t * r * pr * (1.0 - ts) ;
375         p->dfds[2] = t * r * pr * (1.0 + ts) ;
376         p->dfds[4] = -t * r * mr * (1.0 + ts) ;
377         p->dfds[6] = t * r * mr * (1.0 - ts) ;
378         p->dfds[1] = -s * r * pr ;
```

```
379         p->dfds[3] = 0.5 * pmr * (1.0 + ts) ;
380         p->dfds[5] = s * r * mr ;
381         p->dfds[7] = -0.5 * pmr * (1.0 - ts) ;
382         p->dfds[8] = -2.0 * s * pmr ;
383
384         return(0) ;
385   }
386   int     shap_2s3(x,p)
387   struct gausspts             *x ;
388   struct shapefuncs    *p ;
389
390   {
391         double r,s,mr,ms,pr,ps,m3r,p3r,m3s,p3s,r2,mr2,s2,ms2,frs,t,nt ;
392         double temp, mr2p3r,mr2m3r,ms2p3s,ms2m3s ;
393
394         r = x->x ;
395         s = x->y ;
396         if( (r<-1.00001) || (r>1.00001) || (s<-1.00001) || (s>1.00001) )
397               return(12) ;
398
399         t = 0.03125 ;
400         nt = 0.28125 ;
401         p3r = 1.0 + 3.0 * r ;
402         m3r = 1.0 - 3.0 * r ;
403         p3s = 1.0 + 3.0 * s ;
404         m3s = 1.0 - 3.0 * s ;
405         mr = 1.0 - r ;
406         ms = 1.0 - s ;
407         pr = 1.0 + r ;
408         ps = 1.0 + s ;
409         r2 = r * r ;
410         s2 = s * s ;
411         mr2 = nt * (1.0 - r2) ;
412         ms2 = nt * (1.0 - s2) ;
413         frs = t * (-10 + 9 * (r2 + s2)) ;
414         ms2m3s = ms2 * m3s ;
415         ms2p3s = ms2 * p3s ;
416         mr2m3r = mr2 * m3r ;
417         mr2p3r = mr2 * p3r ;
418
419         p->dof = 12 ;
420
421         p->f[0] = pr * ms * frs ;
422         p->f[3] = pr * ps * frs ;
423         p->f[6] = mr * ps * frs ;
424         p->f[9] = mr * ms * frs ;
425         p->f[1] = pr * ms2m3s ;
426         p->f[2] = pr * ms2p3s ;
427         p->f[4] = ps * mr2p3r ;
428         p->f[5] = ps * mr2m3r ;
429         p->f[7] = mr * ms2p3s ;
430         p->f[8] = mr * ms2m3s ;
431         p->f[10] = ms * mr2m3r ;
432         p->f[11] = ms * mr2p3r ;
433
434         if(x->w < 0.0)
435               return(0) ;
436
437         nt *= 2.0 ;
438         r *= nt ;
439         s *= nt ;
440         m3s *= -s ;
441         p3s *= -s ;
```

```
442         m3r *= -r ;
443         p3r *= -r ;
444         ms2 *= 3.0 ;
445         mr2 *= 3.0 ;
446
447         temp = frs + pr * r ;
448         p->dfdr[0] = ms * temp ;
449         p->dfdr[3] = ps * temp ;
450         temp = mr * r - frs ;
451         p->dfdr[6] = ps * temp ;
452         p->dfdr[9] = ms * temp ;
453
454         p->dfdr[1] =  ms2m3s ;
455         p->dfdr[2] =  ms2p3s ;
456         p->dfdr[4] =  ps * (p3r + mr2) ;
457         p->dfdr[5] =  ps * (m3r - mr2) ;
458         p->dfdr[7] = -ms2p3s ;
459         p->dfdr[8] = -ms2m3s ;
460         p->dfdr[10] =  ms * (m3r - mr2) ;
461         p->dfdr[11] =  ms * (p3r + mr2) ;
462
463         temp = frs + ps * s ;
464         p->dfds[3] = pr * temp ;
465         p->dfds[6] = mr * temp ;
466         temp = ms * s - frs ;
467         p->dfds[0] = pr * temp ;
468         p->dfds[9] = mr * temp ;
469
470         p->dfds[1] =  pr * (m3s - ms2) ;
471         p->dfds[2] =  pr * (p3s + ms2) ;
472         p->dfds[4] =  mr2p3r ;
473         p->dfds[5] =  mr2m3r ;
474         p->dfds[7] =  mr * (p3s + ms2) ;
475         p->dfds[8] =  mr * (m3s - ms2) ;
476         p->dfds[10] =  -mr2m3r ;
477         p->dfds[11] =  -mr2p3r ;
478
479         return(0) ;
480 }
481 int     shap_ls1(x,p)
482 struct gausspts             *x ;
483 struct shapefuncs    *p ;
484
485 {
486         double r,mr,pr,t ;
487
488         r = x->x ;
489         if( (r<-1.0) || (r>1.0) ) return(2) ;
490
491         t = 0.5 ;
492         mr = 1.0 - r ;
493         pr = 1.0 + r ;
494
495         p->dof = 2 ;
496
497         p->f[0] =  t * mr ;
498         p->f[1] =  t * pr ;
499
500         if(x->w < 0.0)
501                 return(0) ;
502
503         p->dfdr[0] = -0.5 ;
504         p->dfdr[1] =  0.5 ;
```

```
505
506         return(0) ;
507     }
508     int     shap_ls2(x,p)
509     struct gausspts             *x ;
510     struct shapefuncs    *p ;
511
512     {
513         double r,mr,pr,t ;
514
515         r = x->x ;
516         if( (r<-1.0) || (r>1.0) ) return(3) ;
517
518         t = 0.5 ;
519         mr = 1.0 - r ;
520         pr = 1.0 + r ;
521
522         p->dof = 3 ;
523
524         p->f[0] = -t * r * mr ;
525         p->f[1] = pr * mr ;
526         p->f[2] = t * pr * r ;
527
528         if(x->w < 0.0)
529             return(0) ;
530
531         p->dfdr[0] = -t * (1.0 - 2.0*r) ;
532         p->dfdr[1] = -2.0 * r  ;
533         p->dfdr[2] = t * (1.0 + 2.0*r) ;
534
535         return(0) ;
536     }
537     int     shap_ls3(x,p)
538     struct gausspts             *x ;
539     struct shapefuncs    *p ;
540
541     {
542         double r,rm,rp,r3m,r3p,t ;
543
544         r = x->x ;
545         if( (r<-1.0) || (r>1.0) ) return(4) ;
546
547         t = 9.0/144.0  ;
548         rm = r - 1.0 ;
549         rp = 1.0 + r ;
550         r3m = 3.0 * r - 1.0 ;
551         r3p = 3.0 * r + 1.0 ;
552
553         p->dof = 4 ;
554
555         p->f[0] = -t * rm * r3p * r3m ;
556         p->f[1] = 9.0 * t * rp * rm * r3m ;
557         p->f[2] = -9.0 * t * rp * rm * r3p ;
558         p->f[3] = t * rp * r3p * r3m ;
559
560         if(x->w < 0.0)
561             return(0) ;
562
563         p->dfdr[0] = -t * ( r3p * r3m + 3.0*rm*r3m + 3.0*rm*r3p) ;
564         p->dfdr[1] = 9.0 * t * ( 3.0*rp * rm + rm*r3m + rm*r3p)  ;
565         p->dfdr[2] = -9.0 * t * ( 3.0*rp * rm + rm*r3p + rp*r3p) ;
566         p->dfdr[3] = t * ( r3p * r3m + 3.0*rm*r3m + 3.0*rm*r3p) ;
567
```

```
568         return(0) ;
569     }
570     int     shap_pt(x,p)
571     struct gausspts             *x ;
572     struct shapefuncs    *p ;
573
574     {
575         double r;
576
577         r = x->x ;
578         if( (r<-1.0) || (r>1.0) ) return(1) ;
579
580         p->dof = 1 ;
581         p->f[0] =  1 ;
582         p->dfdr[0] = 0.0 ;
583
584         return(0) ;
585     }
586     int     hermite(x,p)
587     struct gausspts             *x ;
588     struct shapefuncs    *p ;
589
590     {
591     /*
592         double r,s,mr,ms,pr,ps,t,tr,ts ;
593
594         r = x->x ;
595         s = x->y ;
596         if( (r<-1.00001) || (r>1.00001) || (s<-1.00001) || (s>1.00001) )
597               return(4) ;
598
599         t = 1.0/16.0 ;
600         tr = 2.0 * r ;
601         ts = 2.0 * s ;
602         mr = r - 1.0 ;
603         ms = s - 1.0 ;
604         pr = 1.0 + r ;
605         ps = 1.0 + s ;
606
607         p->dof = 12 ;
608
609         p->f[0] = t * pr * pr * (r - 2.0) * ms * ms * (-s - 2.0) ;
610         p->f[3] = t *-pr * pr * (r - 2.0) * ps * ps * (s - 2.0) ;
611         p->f[6] = t * mr * mr * (-r - 2.0) * ps * ps * (s - 2.0) ;
612         p->f[9] = t * mr * mr * (-r - 2.0) * ms * ms * (-s - 2.0) ;
613         p->f[1] = -t * pr * pr * (r - 1.0) * ms * ms * (-s - 2.0) ;
614         p->f[4] = -t * pr * pr * (r - 1.0) * ps * ps * (s - 2.0) ;
615         p->f[7] = t * mr * mr * (-r - 1.0) * ps * ps * (s - 2.0) ;
616         p->f[10] = t * mr * mr * (-r - 1.0) * ms * ms * (-s - 2.0) ;
617         p->f[2] = t * pr * pr * (r - 2.0) * ms * ms * (-s - 1.0) ;
618         p->f[5] = -t * pr * pr * (r - 2.0) * ps * ps * (s - 1.0) ;
619         p->f[8] = -t * mr * mr * (-r - 2.0) * ps * ps * (s - 1.0) ;
620         p->f[11] = t * mr * mr * (-r - 2.0) * ms * ms * (-s - 1.0) ;
621
622         if(x->w < 0.0)
623               return(0) ;
624
625         p->dfdr[0] = t * pr * (3*r - 3.0) * ms * ms * (-s - 2.0) ;
626         p->dfdr[3] = t * pr * (3*r - 3.0) * ps * ps * (s - 2.0) ;
627         p->dfdr[6] = t * mr * (-3*r - 3.0) * ps * ps * (s - 2.0) ;
628         p->dfdr[9] = t * mr * (-3*r - 3.0) * ms * ms * (-s - 2.0) ;
629         p->dfdr[1] = -t * pr * (3*r - 1.0) * ms * ms * (-s - 2.0) ;
630         p->dfdr[4] = -t * pr * (3*r - 1.0) * ps * ps * (s - 2.0) ;
```

```
631         p->dfdr[7] = t * mr * (-3*r - 1.0) * ps * ps * (s - 2.0) ;
632         p->dfdr[10] = t * mr * (-3*r - 1.0) * ms * ms * (-s - 2.0) ;
633         p->dfdr[2] = t * pr * (3*r - 3.0) * ms * ms * (-s - 1.0) ;
634         p->dfdr[5] = -t * pr * (3*r - 3.0) * ps * ps * (s - 1.0) ;
635         p->dfdr[8] = -t * mr * (-3*r - 3.0) * ps * ps * (s - 1.0) ;
636         p->dfdr[11] = t * mr * (-3*r - 3.0) * ms * ms * (-s - 1.0) ;
637
638         p->dfds[0] = t * pr * pr * (r - 2.0) * ms * (-3*s - 3.0) ;
639         p->dfds[3] = t * pr * pr * (r - 2.0) * ps * (3*s - 3.0) ;
640         p->dfds[6] = t * mr * mr * (-r - 2.0) * ps * (3*s - 3.0) ;
641         p->dfds[9] = t * mr * mr * (-r - 2.0) * ms * (-3*s - 3.0) ;
642         p->dfds[1] = -t * pr * pr * (r - 1.0) * ms * (-3*s - 3.0) ;
643         p->dfds[4] = -t * pr * pr * (r - 1.0) * ps * (3*s - 3.0) ;
644         p->dfds[7] = t * mr * mr * (-r - 1.0) * ps * (3*s - 3.0) ;
645         p->dfds[10] = t * mr * mr * (-r - 1.0) * ms * (-3*s - 3.0) ;
646         p->dfds[2] = t * pr * pr * (r - 2.0) * ms * (-3*s - 1.0) ;
647         p->dfds[5] = -t * pr * pr * (r - 2.0) * ps * (3*s - 1.0) ;
648         p->dfds[8] = -t * mr * mr * (-r - 2.0) * ps * (3*s - 1.0) ;
649         p->dfds[11] = t * mr * mr * (-r - 2.0) * ms * (-3*s - 1.0) ;
650
651         */
652
653         return(0) ;
654 }
655
656
657
658 int     shqu_211(x,p)
659 struct gausspts             *x ;
660 struct shapefuncs    *p ;
661
662 {
663         double r,s,mr,ms ;
664
665         r = x->x ;
666         s = x->y ;
667         if( (r<-1.00001) || (r>1.00001) || (s<-1.00001) || (s>1.00001) )
668 return(4) ;
669
670         mr = 1.0 - r ;
671         ms = 1.0 - s ;
672
673         p->dof = 4 ;
674
675         p->f[0] = r * ms ;
676         p->f[1] = r * s ;
677         p->f[2] = mr * s ;
678         p->f[3] = mr * ms ;
679
680         if(x->w < 0.0)
681                 return(0) ;
682
683         p->dfdr[0] =  ms ;
684         p->dfdr[1] =  s ;
685         p->dfdr[2] = -s ;
686         p->dfdr[3] = -ms ;
687
688         p->dfds[0] = -r ;
689         p->dfds[1] =  r ;
690         p->dfds[2] =  mr ;
691         p->dfds[3] = -mr ;
692
693         return(0) ;
```

```
694   }
695
696
697   int     shqu9_212(x,p)
698   struct gausspts             *x ;
699   struct shapefuncs     *p ;
700
701   {
702         double r,s,mr,ms,pr,ps,t,tr,ts,pmr,pms ;
703
704         r = x->x ;
705         s = x->y ;
706         if( (r<-1.00001) || (r>1.00001) || (s<-1.00001) || (s>1.00001) )
707   return(9) ;
708
709         t = 0.25 ;
710         tr = 2.0 * r ;
711         ts = 2.0 * s ;
712         mr = 1.0 - r ;
713         ms = 1.0 - s ;
714         pr = 1.0 + r ;
715         ps = 1.0 + s ;
716         pmr = pr * mr ;
717         pms = ps * ms ;
718
719         p->dof = 9 ;
720
721         p->f[8] = -t * pr * ms * r * s ;
722         p->f[1] = t * pr * ps * r * s ;
723         p->f[4] = -t * r * mr * s * ps ;
724         p->f[6] = t * r * mr * s * ms ;
725         p->f[0] = 0.5 * r * pr * pms ;
726         p->f[2] = 0.5 * pmr * s * ps ;
727         p->f[5] = -0.5 * r * mr * pms ;
728         p->f[7] = -0.5 * pmr * s * ms ;
729         p->f[3] = pmr * pms ;
730
731         if(x->w < 0.0)
732               return(0) ;
733
734         p->dfdr[8] = -t * s * ms * (tr + 1.0) ;
735         p->dfdr[1] = t * s * ps * (1.0 + tr) ;
736         p->dfdr[4] = -t * s * ps * (1.0 - tr) ;
737         p->dfdr[6] = t * s * ms * (1.0 - tr) ;
738         p->dfdr[0] =  0.5 * pms * (1.0 + tr) ;
739         p->dfdr[2] = -r * s * ps ;
740         p->dfdr[5] = -0.5 * (1.0 - tr) * pms ;
741         p->dfdr[7] =  r * s * ms ;
742         p->dfdr[3] = -2.0 * r * pms ;
743
744         p->dfds[8] = -t * r * pr * (1.0 - ts) ;
745         p->dfds[1] = t * r * pr * (1.0 + ts) ;
746         p->dfds[4] = -t * r * mr * (1.0 + ts) ;
747         p->dfds[6] = t * r * mr * (1.0 - ts) ;
748         p->dfds[0] = -s * r * pr ;
749         p->dfds[2] = 0.5 * pmr * (1.0 + ts) ;
750         p->dfds[5] = s * r * mr ;
751         p->dfds[7] = -0.5 * pmr * (1.0 - ts) ;
752         p->dfds[3] = -2.0 * s * pmr ;
753
754         return(0) ;
755   }
756
```

```
757
758   int     shqu8_212(x,p)
759   struct gausspts             *x ;
760   struct shapefuncs     *p ;
761
762   {
763         double r,s,mr,ms,pr,ps,t,tr,ts,pmr,pms ;
764
765         r = x->x ;
766         s = x->y ;
767         if( (r<-1.00001) || (r>1.00001) || (s<-1.00001) || (s>1.00001) )
768   return(8) ;
769
770         t = 0.25 ;
771         tr = 2.0 * r ;
772         ts = 2.0 * s ;
773         mr = 1.0 - r ;
774         ms = 1.0 - s ;
775         pr = 1.0 + r ;
776         ps = 1.0 + s ;
777         pmr = pr * mr ;
778         pms = ps * ms ;
779
780         p->dof = 8 ;
781
782         p->f[7] = -0.5 * ms * r * s ;
783         p->f[1] = t * pr * ps * r * s ;
784         p->f[4] = -0.5 * r * mr * s ;
785         p->f[0] = 0.5 * r * pr * pms ;
786         p->f[2] = 0.5 * pmr * s * ps ;
787         p->f[5] = -0.5 * r * mr * ms ;
788         p->f[6] = -0.5 * mr * s * ms ;
789         p->f[3] = pmr * pms ;
790
791         if(x->w < 0.0)
792               return(0) ;
793
794         p->dfdr[7] = -0.5 * ms * s  ;
795         p->dfdr[1] = t * s * ps * (1.0 + tr) ;
796         p->dfdr[4] = -0.5 * s * (1.0 - tr) ;
797         p->dfdr[0] =  0.5 * pms * (1.0 + tr) ;
798         p->dfdr[2] = -r * s * ps ;
799         p->dfdr[5] = -0.5 * (1.0 - tr) * ms ;
800         p->dfdr[6] =  0.5 * s * ms ;
801         p->dfdr[3] = -2.0 * r * pms ;
802
803         p->dfds[7] = 0.5 * r * (1.0 - ts) ;
804         p->dfds[1] = t * r * pr * (1.0 + ts) ;
805         p->dfds[4] = -0.5 * r * mr ;
806         p->dfds[0] = -s * r * pr ;
807         p->dfds[2] = 0.5 * pmr * (1.0 + ts) ;
808         p->dfds[5] = 0.5 * r * mr ;
809         p->dfds[6] = -0.5 * mr * (1.0 - ts) ;
810         p->dfds[3] = -2.0 * s * pmr ;
811
812         return(0) ;
813   }
814
815   int     shqu6_212(x,p)
816   struct gausspts             *x ;
817   struct shapefuncs     *p ;
818
819   {
```

```
820         double r,s,mr,ms,pr,ps,t,tr,ts,pmr,pms ;
821
822         r = x->x ;
823         s = x->y ;
824         if( (r<-1.00001) || (r>1.00001) || (s<-1.00001) || (s>1.00001) )
825               return(6) ;
826
827         t = 0.25 ;
828         tr = 2.0 * r ;
829         ts = 2.0 * s ;
830         mr = 1.0 - r ;
831         ms = 1.0 - s ;
832         pr = 1.0 + r ;
833         ps = 1.0 + s ;
834         pmr = pr * mr ;
835         pms = ps * ms ;
836
837         p->dof = 6 ;
838
839         p->f[1] = 0.5 * pr * r * s ;
840         p->f[4] = -0.5 * r * mr * s ;
841         p->f[0] = 0.5 * r * pr * ms ;
842         p->f[2] = pmr * s  ;
843         p->f[5] = -0.5 * r * mr * ms ;
844         p->f[3] = pmr * ms ;
845
846         if(x->w < 0.0)
847               return(0) ;
848
849         p->dfdr[1] =  0.5 * s * (1.0 + tr) ;
850         p->dfdr[4] = -0.5 * s * (1.0 - tr) ;
851         p->dfdr[0] =  0.5 * ms * (1.0 + tr) ;
852         p->dfdr[2] = -2.0 * r * s ;
853         p->dfdr[5] = -0.5 * (1.0 - tr) * ms ;
854         p->dfdr[3] = -2.0 * r * ms ;
855
856         p->dfds[1] = 0.5 * pr * r ;
857         p->dfds[4] = -0.5 * r * mr ;
858         p->dfds[0] = -0.5 * r * pr ;
859         p->dfds[2] = pmr ;
860         p->dfds[5] = 0.5 * r * mr ;
861         p->dfds[3] = -pmr ;
862
863         return(0) ;
864   }
865
866
867
```

```
1   /*      the shape functions for the cubic upwinding functions   239          */
2   #include "fe.h"
3
4
5   extern struct control  N ;
6
7
8
9   int     shcu2_2x1(x,p)
10  struct gausspts      *x;
11  struct shapefuncs    *p;
12
13  {
14          double r, s, t, rm, sm, rp, sp, rp3;
15
16          r = x->x ;
17          s = x->y ;
18          if ( (r<-3.00001) || (r>1.00001) || (s<-1.00001) || (s>1.00001) )
19  return(6) ;
20
21          t = 0.125 ;
22          rm = 1.0 - r ;
23          rp = 1.0 + r ;
24          sp = 1.0 + s ;
25          sm = 1.0 - s ;
26          rp3 = 3.0 + r ;
27
28          p->dof = 6 ;
29
30          p->f[3] = t * rp3 * sm * rm ;
31          p->f[0] = t / 2.0 * rp3 * rp * sm ;
32          p->f[1] = t / 2.0 * rp * sp * rp3 ;
33          p->f[2] = t * rm * sp * rp3 ;
34          p->f[4] = t / -2.0 * rp * sp * rm ;
35          p->f[5] = t / -2.0 * rp * sm * rm ;
36
37          if (x->w < 0.0)
38                  return(0) ;
39
40          p->dfdr[3] = -2.0 * t * sm * rp ;
41          p->dfdr[0] = t * sm * ( r + 2.0 ) ;
42          p->dfdr[1] = t * sp * ( r + 2.0 ) ;
43          p->dfdr[2] = -2.0 * t * sp * rp ;
44          p->dfdr[4] = t * sp * r ;
45          p->dfdr[5] = t * sm * r ;
46
47          p->dfds[3] = -t * rp3 * rm ;
48          p->dfds[0] = -t / 2.0 * rp3 * rp ;
49          p->dfds[1] = t / 2.0 * rp3 * rp ;
50          p->dfds[2] = t * rp3 * rm ;
51          p->dfds[4] = -t / 2.0 * rp * rm ;
52          p->dfds[5] = t / 2.0 * rp * rm ;
53
54          return (0);
55  }
56
57
58  int     shcu3_2x2(x,p)
59  struct gausspts      *x;
60  struct shapefuncs    *p;
61
62  {
63          double r, s, t, rm, sm, rp, sp, rp3, sp3;
```

```
64
65          r = x->x ;
66          s = x->y ;
67          if ( (r<-3.00001) || (r>1.00001) || (s<-3.00001) || (s>1.00001) )
68      return(8) ;
69
70          t = 0.0625 ;
71          rm = 1.0 - r ;
72          rp = 1.0 + r ;
73          sp = 1.0 + s ;
74          sm = 1.0 - s ;
75          rp3 = 3.0 + r ;
76          sp3 = 3.0 + s ;
77
78          p->dof = 8 ;
79
80          p->f[3] = t * rp3 * sp * rm * sp3 ;
81          p->f[0] = t / 2.0 * rp3 * rp * sm * sp3 ;
82          p->f[1] = t / 4.0 * rp * sp * rp3 * sp3 ;
83          p->f[2] = t / 2.0 * rm * sp * rp3 * sp3 ;
84          p->f[4] = -t * rp * sp * rm ;
85          p->f[5] = -t * rp * sm * rm ;
86          p->f[6] = -t * sm * rm * sp ;
87          p->f[7] = -t * rp * sm * sp ;
88
89
90          if (x->w < 0.0)
91                return(0) ;
92
93          p->dfdr[3] = -2.0 * t * sm * rp * sp3;
94          p->dfdr[0] = t * sm * ( r + 2.0 ) * sp3;
95          p->dfdr[1] = t / 2.0 * sp * ( r + 2.0 ) * sp3;
96          p->dfdr[2] = -t * sp * rp * sp3;
97          p->dfdr[4] = 2. * t * sp * r ;
98          p->dfdr[5] = 2. * t * sm * r ;
99          p->dfdr[6] = t * sm * sp;
100         p->dfdr[7] = -t * sm * sp;
101
102
103         p->dfds[3] = -2.0 * t * sp * rp3 * rm ;
104         p->dfds[0] = -t * sp * rp3 * rp ;
105         p->dfds[1] = t / 2.0 * ( s + 2.0 ) * rp3 * rp ;
106         p->dfds[2] = t * ( s + 2.0 ) * rp3 * rm ;
107         p->dfds[4] = -t * rp * rm ;
108         p->dfds[5] = t * rp * rm ;
109         p->dfds[6] = t * 2.0 * s * rm ;
110         p->dfds[7] = t * 2.0 * s * rp ;
111
112         return (0);
113     }
114
115     int     shcu4_2x2(x,p)
116     struct gausspts       *x;
117     struct shapefuncs     *p;
118
119     {
120         double r, s, t, rm, sm, rp, sp, rp3, sp3;
121
122         r = x->x ;
123         s = x->y ;
124         if ( (r<-3.00001) || (r>1.00001) || (s<-3.00001) || (s>1.00001) )
125     return(9) ;
126
```

```
127         t = 0.0625 ;
128         rm = 1.0 - r ;
129         rp = 1.0 + r ;
130         sp = 1.0 + s ;
131         sm = 1.0 - s ;
132         rp3 = 3.0 + r ;
133         sp3 = 3.0 + s ;
134
135         p->dof = 9 ;
136
137         p->f[3] = t * rp3 * sm * rm * sp3 ;
138         p->f[0] = t / 2.0 * rp3 * rp * sm * sp3 ;
139         p->f[1] = t / 4.0 * rp * sp * rp3 * sp3 ;
140         p->f[2] = t / 2.0 * rm * sp * rp3 * sp3 ;
141         p->f[4] = t / -4.0 * rp * sp * rm * sp3 ;
142         p->f[5] = t / -2.0 * rp * sm * rm * sp3 ;
143         p->f[7] = t / -2.0 * rp3 * sm * rm * sp ;
144         p->f[8] = t / -4.0 * rp * sm * rp3 * sp ;
145         p->f[6] = t / 4.0 * rp * sm * rm * sp ;
146
147
148         if (x->w < 0.0)
149               return(0) ;
150
151         p->dfdr[3] = -2.0 * t * sm * rp * sp3;
152         p->dfdr[0] = t * sm * ( r + 2.0 ) * sp3;
153         p->dfdr[1] = t / 2.0 * sp * ( r + 2.0 ) * sp3;
154         p->dfdr[2] = -t * sp * rp * sp3;
155         p->dfdr[4] = t / 2.0 * sp * r * sp3;
156         p->dfdr[5] = t * sm * r * sp3;
157         p->dfdr[7] = t * sm * rp * sp;
158         p->dfdr[8] = -t / 2.0 * sm * ( r + 2.0 ) * sp;
159         p->dfdr[6] = -t / 2.0 * sm * r * sp;
160
161
162         p->dfds[3] = -2.0 * t * sp * rp3 * rm ;
163         p->dfds[0] = -t * sp * rp3 * rp ;
164         p->dfds[1] = t / 2.0 * ( s + 2.0 ) * rp3 * rp ;
165         p->dfds[2] = t * ( s + 2.0 ) * rp3 * rm ;
166         p->dfds[4] = -t / 2.0 * ( s + 2.0 ) * rp * rm ;
167         p->dfds[5] = t * sp * rp * rm ;
168         p->dfds[7] = t * s * rp3 * rm ;
169         p->dfds[8] = t / 2.0 * s * rp * rp3 ;
170         p->dfds[6] = -t / 2.0 * s * rp * rm ;
171
172         return (0);
173   }
174
175   int     shcu3_3x1(x,p)
176   struct gausspts       *x;
177   struct shapefuncs     *p;
178
179   {
180         double r, s, t, rm, sm, rp, sp, rp3, rp5, r2p8 ;
181
182         r = x->x ;
183         s = x->y ;
184         if ( (r<-5.00001) || (r>1.00001) || (s<-1.00001) || (s>1.00001) )
185   return(8) ;
186
187         t = 1.0 / 32. ;
188         rm = 1.0 - r ;
189         rp = 1.0 + r ;
```

```
190         sp = 1.0 + s ;
191         sm = 1.0 - s ;
192         rp3 = 3.0 + r ;
193         rp5 = 5.0 + r ;
194
195         p->dof = 8 ;
196
197         p->f[3] = t * rp3 * sm * rm * rp5 ;
198         p->f[0] = t / 3.0 * rp3 * rp * sm * rp5 ;
199         p->f[1] = t / 3.0 * rp * sp * rp3 * rp5 ;
200         p->f[2] = t * rm * sp * rp3 * rp5 ;
201         p->f[4] = -t * rp * sp * rm * rp5 ;
202         p->f[5] = t / 3.0 * rp * sp * rm * rp3 ;
203         p->f[6] = t / 3.0 * rp * sm * rm * rp3 ;
204         p->f[7] = -t * rp * sm * rm * rp5 ;
205
206         if (x->w < 0.0)
207               return(0) ;
208
209         r2p8 = 2.0 * r + 8.0 ;
210
211         p->dfdr[3] = t * sm * ( (-rp5 * rp3) + rm * r2p8 ) ;
212         p->dfdr[0] = t / 3.0 * sm * ( ( rp5 * rp3) + rp * r2p8 ) ;
213         p->dfdr[1] = t / 3.0 * sp * ( ( rp5 * rp3) + rp * r2p8 ) ;
214         p->dfdr[2] = t * sp * ( (-rp5 * rp3) + rm * r2p8 ) ;
215         p->dfdr[4] = -t * sp * ( -2.0 * r * rp5 + rp * rm ) ;
216         p->dfdr[5] = t / 3.0 * sp * ( -2.0 * r * rp3 + rp * rm ) ;
217         p->dfdr[6] = t / 3.0 * sm * ( -2.0 * r * rp3 + rp * rm ) ;
218         p->dfdr[7] = -t * sm * ( -2.0 * r * rp5 + rp * rm ) ;
219
220         p->dfds[3] = -t * rp3 * rm * rp5 ;
221         p->dfds[0] = -t / 3.0 * rp3 * rp * rp5 ;
222         p->dfds[1] = t / 3.0 * rp * rp3 * rp5 ;
223         p->dfds[2] = t * rm * rp3 * rp5 ;
224         p->dfds[4] = -t * rp * rm * rp5 ;
225         p->dfds[5] = t / 3.0 * rp * rm * rp3 ;
226         p->dfds[6] = -t / 3.0 * rp * rm * rp3 ;
227         p->dfds[7] = t * rp * rm * rp5 ;
228
229         return (0);
230   }
231
232   int     shcu4_3x2(x,p)
233   struct gausspts      *x;
234   struct shapefuncs    *p;
235
236   {
237         double r, s, t, rm, sm, rp, sp, rp3, rp5, r2p8, sp3 ;
238
239         r = x->x ;
240         s = x->y ;
241         if ( (r<-5.00001) || (r>1.00001) || (s<-3.00001) || (s>1.00001) )
242   return(10) ;
243
244         t = 1.0 / 64. ;
245         rm = 1.0 - r ;
246         rp = 1.0 + r ;
247         sp = 1.0 + s ;
248         sm = 1.0 - s ;
249         rp3 = 3.0 + r ;
250         rp5 = 5.0 + r ;
251         sp3 = 3.0 + s ;
252
```

```
253         p->dof = 10 ;
254
255         p->f[3] = t * rp3 * sm * rm * rp5 * sp3 ;
256         p->f[0] = t / 3.0 * rp3 * rp * sm * rp5 * sp3 ;
257         p->f[1] = t / 6.0 * rp * sp * rp3 * rp5 * sp3 ;
258         p->f[2] = t / 2.0 * rm * sp * rp3 * rp5 * sp3 ;
259         p->f[4] = -2.0 * t * rp * sp * rm * rp5 ;
260         p->f[5] = 2.0 * t / 3.0 * rp * sp * rm * rp3 ;
261         p->f[6] = 2.0 * t / 3.0 * rp * sm * rm * rp3 ;
262         p->f[7] = -2.0 * t * rp * sm * rm * rp5 ;
263         p->f[8] = -4.0 * t * sp * rm * sm ;
264         p->f[9] = -4.0 * t * sp * rp * sm ;
265
266         if (x->w < 0.0)
267               return(0) ;
268
269         r2p8 = 2.0 * r + 8.0 ;
270
271         p->dfdr[3] = t * sm * sp3 * ( (-rp5 * rp3 ) + rm * r2p8 ) ;
272         p->dfdr[0] = t / 3.0 * sm * sp3 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
273         p->dfdr[1] = t / 6.0 * sp * sp3 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
274         p->dfdr[2] = t / 2.0 * sp * sp3 * ( (-rp5 * rp3) + rm * r2p8 ) ;
275         p->dfdr[4] = -2.0 * t * sp * ( -2.0 * r * rp5 + rp * rm ) ;
276         p->dfdr[5] = 2.0 * t / 3.0 * sp * ( -2.0 * r * rp3 + rp * rm ) ;
277         p->dfdr[6] = 2.0 * t / 3.0 * sm * ( -2.0 * r * rp3 + rp * rm ) ;
278         p->dfdr[7] = -2.0 * t * sm * ( -2.0 * r * rp5 + rp * rm ) ;
279         p->dfdr[8] = 4.0 * t * sm * sp ;
280         p->dfdr[9] = -4.0 * t * sm * sp ;
281
282         p->dfds[3] = -t * 2.0 * rp3 * sp * rm * rp5 ;
283         p->dfds[0] = -t * 2.0 / 3.0 * rp3 * rp * sp * rp5 ;
284         p->dfds[1] = t / 3.0 * rp * rp3 * rp5 * ( s + 2.0 ) ;
285         p->dfds[2] = t * rm * rp3 * rp5 * ( s + 2.0 ) ;
286         p->dfds[4] = -2.0 * t * rp * rm * rp5 ;
287         p->dfds[5] = 2.0 * t / 3.0 * rp * rm * rp3 ;
288         p->dfds[6] = -2.0 * t / 3.0 * rp * rm * rp3 ;
289         p->dfds[7] = 2.0 * t * rp * rm * rp5 ;
290         p->dfds[8] = 8.0 * t * s * rm ;
291         p->dfds[9] = 8.0 * t * s * rp ;
292
293         return (0);
294   }
295
296   int     sh4_3x2b(x,p)
297   struct gausspts       *x;
298   struct shapefuncs     *p;
299
300   {
301         double r, s, t, rm, sm, rp, sp, rp3, rp5, r2p8, sp3 ;
302
303         r = x->y ;
304         s = x->x ;
305         if ( (r<-5.00001) || (r>1.00001) || (s<-3.00001) || (s>1.00001) )
306   return(10) ;
307
308         t = 1.0 / 64. ;
309         rm = 1.0 - r ;
310         rp = 1.0 + r ;
311         sp = 1.0 + s ;
312         sm = 1.0 - s ;
313         rp3 = 3.0 + r ;
314         rp5 = 5.0 + r ;
315         sp3 = 3.0 + s ;
```

```
316
317         p->dof = 10 ;
318
319         p->f[3] = t * rp3 * sm * rm * rp5 * sp3 ;
320         p->f[2] = t / 3.0 * rp3 * rp * sm * rp5 * sp3 ;
321         p->f[1] = t / 6.0 * rp * sp * rp3 * rp5 * sp3 ;
322         p->f[0] = t / 2.0 * rm * sp * rp3 * rp5 * sp3 ;
323         p->f[4] = -2.0 * t * rp * sp * rm * rp5 ;
324         p->f[5] = 2.0 * t / 3.0 * rp * sp * rm * rp3 ;
325         p->f[6] = 2.0 * t / 3.0 * rp * sm * rm * rp3 ;
326         p->f[7] = -2.0 * t * rp * sm * rm * rp5 ;
327         p->f[8] = -4.0 * t * sp * rm * sm ;
328         p->f[9] = -4.0 * t * sp * rp * sm ;
329
330         if (x->w < 0.0)
331               return(0) ;
332
333         r2p8 = 2.0 * r + 8.0 ;
334
335         p->dfds[3] = t * sm * sp3 * ( (-rp5 * rp3 ) + rm * r2p8 ) ;
336         p->dfds[2] = t / 3.0 * sm * sp3 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
337         p->dfds[1] = t / 6.0 * sp * sp3 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
338         p->dfds[0] = t / 2.0 * sp * sp3 * ( (-rp5 * rp3) + rm * r2p8 ) ;
339         p->dfds[4] = -2.0 * t * sp * ( -2.0 * r * rp5 + rp * rm ) ;
340         p->dfds[5] = 2.0 * t / 3.0 * sp * ( -2.0 * r * rp3 + rp * rm ) ;
341         p->dfds[6] = 2.0 * t / 3.0 * sm * ( -2.0 * r * rp3 + rp * rm ) ;
342         p->dfds[7] = -2.0 * t * sm * ( -2.0 * r * rp5 + rp * rm ) ;
343         p->dfds[8] = 4.0 * t * sm * sp ;
344         p->dfds[9] = -4.0 * t * sm * sp ;
345
346         p->dfdr[3] = -t * 2.0 * rp3 * sp * rm * rp5 ;
347         p->dfdr[2] = -t * 2.0 / 3.0 * rp3 * rp * sp * rp5 ;
348         p->dfdr[1] = t / 3.0 * rp * rp3 * rp5 * ( s + 2.0 ) ;
349         p->dfdr[0] = t * rm * rp3 * rp5 * ( s + 2.0 ) ;
350         p->dfdr[4] = -2.0 * t * rp * rm * rp5 ;
351         p->dfdr[5] = 2.0 * t / 3.0 * rp * rm * rp3 ;
352         p->dfdr[6] = -2.0 * t / 3.0 * rp * rm * rp3 ;
353         p->dfdr[7] = 2.0 * t * rp * rm * rp5 ;
354         p->dfdr[8] = 8.0 * t * s * rm ;
355         p->dfdr[9] = 8.0 * t * s * rp ;
356
357         return (0);
358   }
359
360   int     shcu5_3x2(x,p)
361   struct gausspts      *x;
362   struct shapefuncs    *p;
363
364   {
365         double r, s, t, rm, sm, rp, sp, rp3, rp5, r2p8, sp3 ;
366
367         r = x->x ;
368         s = x->y ;
369         if ( (r<-5.00001) || (r>1.00001) || (s<-3.00001) || (s>1.00001) )
370   return(11) ;
371
372         t = 1.0 / 64. ;
373         rm = 1.0 - r ;
374         rp = 1.0 + r ;
375         sp = 1.0 + s ;
376         sm = 1.0 - s ;
377         rp3 = 3.0 + r ;
378         rp5 = 5.0 + r ;
```

```
379         sp3 = 3.0 + s ;
380
381         p->dof = 11 ;
382
383         p->f[3] = t * rp3 * sm * rm * rp5 * sp3 ;
384         p->f[0] = t / 3.0 * rp3 * rp * sm * rp5 * sp3 ;
385         p->f[1] = t / 6.0 * rp * sp * rp3 * rp5 * sp3 ;
386         p->f[2] = t / 2.0 * rm * sp * rp3 * rp5 * sp3 ;
387         p->f[4] = -t / 2.0 * rp * sp * rm * rp5 * sp3 ;
388         p->f[5] = 2.0 * t / 3.0 * rp * sp * rm * rp3 ;
389         p->f[6] = 2.0 * t / 3.0 * rp * sm * rm * rp3 ;
390         p->f[7] = -t * rp * sm * rm * rp5 * sp3 ;
391         p->f[8] = t * rp * sp * rm * sm ;
392         p->f[9] = -2.0 * t * rp3 * sp * rm * sm ;
393         p->f[10] = -t * rp3 * sp * rp * sm ;
394
395         if (x->w < 0.0)
396               return(0) ;
397
398         r2p8 = 2.0 * r + 8.0 ;
399
400         p->dfdr[3] = t * sm * sp3 * ( (-rp5 * rp3 ) + rm * r2p8 ) ;
401         p->dfdr[0] = t / 3.0 * sm * sp3 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
402         p->dfdr[1] = t / 6.0 * sp * sp3 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
403         p->dfdr[2] = t / 2.0 * sp * sp3 * ( (-rp5 * rp3) + rm * r2p8 ) ;
404         p->dfdr[4] = -t / 2.0 * sp * sp3 * ( -2.0 * r * rp5 + rp * rm ) ;
405         p->dfdr[5] = 2.0 * t / 3.0 * sp * ( -2.0 * r * rp3 + rp * rm ) ;
406         p->dfdr[6] = 2.0 * t / 3.0 * sm * ( -2.0 * r * rp3 + rp * rm ) ;
407         p->dfdr[7] = -t * sm * sp3 * ( -2.0 * r * rp5 + rp * rm ) ;
408         p->dfdr[8] = t * sp * sm * ( -2.0 * r ) ;
409         p->dfdr[9] = -t * 2.0 * sm * sp * ( -2.0 * rp ) ;
410         p->dfdr[10] = -t * sp * sm * ( 2.0 * r + 4.0 ) ;
411
412         p->dfds[3] = -t * 2.0 * rp3 * sp * rm * rp5 ;
413         p->dfds[0] = -t * 2.0 / 3.0 * rp3 * rp * sp * rp5 ;
414         p->dfds[1] = t / 3.0 * rp * rp3 * rp5 * ( s + 2.0 ) ;
415         p->dfds[2] = t * rm * rp3 * rp5 * ( s + 2.0 ) ;
416         p->dfds[4] = -t * rp * rm * rp5 * ( s + 2.0 ) ;
417         p->dfds[5] = 2.0 * t / 3.0 * rp * rm * rp3 ;
418         p->dfds[6] = -2.0 * t / 3.0 * rp * rm * rp3 ;
419         p->dfds[7] = 2.0 * t * rp * rm * rp5 * sp ;
420         p->dfds[8] = -2.0 * t * rp * s * rm ;
421         p->dfds[9] = 4.0 * t * rp3 * s * rm ;
422         p->dfds[10] = 2.0 * t * rp3 * s * rp ;
423
424         return (0);
425   }
426
427   int     sh5_3x2b(x,p)
428   struct gausspts       *x;
429   struct shapefuncs     *p;
430
431   {
432         double r, s, t, rm, sm, rp, sp, rp3, rp5, r2p8, sp3 ;
433
434         r = x->y ;
435         s = x->x ;
436         if ( (r<-5.00001) || (r>1.00001) || (s<-3.00001) || (s>1.00001) )
437   return(11) ;
438
439         t = 1.0 / 64. ;
440         rm = 1.0 - r ;
441         rp = 1.0 + r ;
```

```
442         sp = 1.0 + s ;
443         sm = 1.0 - s ;
444         rp3 = 3.0 + r ;
445         rp5 = 5.0 + r ;
446         sp3 = 3.0 + s ;
447
448         p->dof = 11 ;
449
450         p->f[3] = t * rp3 * sm * rm * rp5 * sp3 ;
451         p->f[2] = t / 3.0 * rp3 * rp * sm * rp5 * sp3 ;
452         p->f[1] = t / 6.0 * rp * sp * rp3 * rp5 * sp3 ;
453         p->f[0] = t / 2.0 * rm * sp * rp3 * rp5 * sp3 ;
454         p->f[4] = -t / 2.0 * rp * sp * rm * rp5 * sp3 ;
455         p->f[5] = 2.0 * t / 3.0 * rp * sp * rm * rp3 ;
456         p->f[6] = 2.0 * t / 3.0 * rp * sm * rm * rp3 ;
457         p->f[7] = -t * rp * sm * rm * rp5 * sp3 ;
458         p->f[8] = t * rp * sp * rm * sm ;
459         p->f[9] = -2.0 * t * rp3 * sp * rm * sm ;
460         p->f[10] = -t * rp3 * sp * rp * sm ;
461
462         if (x->w < 0.0)
463               return(0) ;
464
465         r2p8 = 2.0 * r + 8.0 ;
466
467         p->dfds[3] = t * sm * sp3 * ( (-rp5 * rp3 ) + rm * r2p8 ) ;
468         p->dfds[2] = t / 3.0 * sm * sp3 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
469         p->dfds[1] = t / 6.0 * sp * sp3 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
470         p->dfds[0] = t / 2.0 * sp * sp3 * ( (-rp5 * rp3) + rm * r2p8 ) ;
471         p->dfds[4] = -t / 2.0 * sp * sp3 * ( -2.0 * r * rp5 + rp * rm ) ;
472         p->dfds[5] = 2.0 * t / 3.0 * sp * ( -2.0 * r * rp3 + rp * rm ) ;
473         p->dfds[6] = 2.0 * t / 3.0 * sm * ( -2.0 * r * rp3 + rp * rm ) ;
474         p->dfds[7] = -t * sm * sp3 * ( -2.0 * r * rp5 + rp * rm ) ;
475         p->dfds[8] = t * sp * sm * ( -2.0 * r ) ;
476         p->dfds[9] = -t * 2.0 * sm * sp * ( -2.0 * rp ) ;
477         p->dfds[10] = -t * sp * sm * ( 2.0 * r + 4.0 ) ;
478
479         p->dfdr[3] = -t * 2.0 * rp3 * sp * rm * rp5 ;
480         p->dfdr[2] = -t * 2.0 / 3.0 * rp3 * rp * sp * rp5 ;
481         p->dfdr[1] = t / 3.0 * rp * rp3 * rp5 * ( s + 2.0 ) ;
482         p->dfdr[0] = t * rm * rp3 * rp5 * ( s + 2.0 ) ;
483         p->dfdr[4] = -t * rp * rm * rp5 * ( s + 2.0 ) ;
484         p->dfdr[5] = 2.0 * t / 3.0 * rp * rm * rp3 ;
485         p->dfdr[6] = -2.0 * t / 3.0 * rp * rm * rp3 ;
486         p->dfdr[7] = 2.0 * t * rp * rm * rp5 * sp ;
487         p->dfdr[8] = -2.0 * t * rp * s * rm ;
488         p->dfdr[9] = 4.0 * t * rp3 * s * rm ;
489         p->dfdr[10] = 2.0 * t * rp3 * s * rp ;
490
491         return (0);
492   }
493
```

```
1   /*     the shape functions for the cubic upwinding functions   239          */
2   #include "fe.h"
3
4
5   extern struct control  N ;
6
7   int     shcu6_3x2(x,p)
8   struct gausspts      *x;
9   struct shapefuncs    *p;
10
11  {
12          double r, s, t, rm, sm, rp, sp, rp3, rp5, r2p8, sp3 ;
13
14          r = x->x ;
15          s = x->y ;
16          if ( (r<-5.00001) || (r>1.00001) || (s<-3.00001) || (s>1.00001) )
17  return(12) ;
18
19          t = 1.0 / 64. ;
20          rm = 1.0 - r ;
21          rp = 1.0 + r ;
22          sp = 1.0 + s ;
23          sm = 1.0 - s ;
24          rp3 = 3.0 + r ;
25          rp5 = 5.0 + r ;
26          sp3 = 3.0 + s ;
27
28          p->dof = 12 ;
29
30          p->f[3] = t * rp3 * sm * rm * rp5 * sp3 ;
31          p->f[0] = t / 3.0 * rp3 * rp * sm * rp5 * sp3 ;
32          p->f[1] = t / 6.0 * rp * sp * rp3 * rp5 * sp3 ;
33          p->f[2] = t / 2.0 * rm * sp * rp3 * rp5 * sp3 ;
34          p->f[4] = -t / 2.0 * rp * sp * rm * rp5 * sp3 ;
35          p->f[5] = t / 6.0 * rp * sp * rm * rp3 * sp3;
36          p->f[6] = t / 3.0 * rp * sm * rm * rp3 * sp3;
37          p->f[7] = -t * rp * sm * rm * rp5 * sp3;
38          p->f[8] = -t / 6.0 * rp3 * sp * rp * rm * sm ;
39          p->f[9] = t / 2.0 * rp * sp * rm * rp5 * sm ;
40          p->f[10] = -t / 2.0 * rp3 * sp * rm * rp5 * sm ;
41          p->f[11] = -t / 6.0 * rp3 * sp * rp * rp5 * sm ;
42
43          if (x->w < 0.0)
44                  return(0) ;
45
46          r2p8 = 2.0 * r + 8.0 ;
47
48          p->dfdr[3] = t * sm * sp3 * ( (-rp5 * rp3 ) + rm * r2p8 ) ;
49          p->dfdr[0] = t / 3.0 * sm * sp3 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
50          p->dfdr[1] = t / 6.0 * sp * sp3 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
51          p->dfdr[2] = t / 2.0 * sp * sp3 * ( (-rp5 * rp3) + rm * r2p8 ) ;
52          p->dfdr[4] = -t / 2.0 * sp * sp3 * ( -2.0 * r * rp5 + rp * rm ) ;
53          p->dfdr[5] = t / 6.0 * sp * sp3 * ( -2.0 * r * rp3 + rp * rm ) ;
54          p->dfdr[6] = t / 3.0 * sm * sp3 * ( -2.0 * r * rp3 + rp * rm ) ;
55          p->dfdr[7] = -t * sm * sp3 * ( -2.0 * r * rp5 + rp * rm ) ;
56          p->dfdr[8] = -t / 6.0 * sp * sm * ( -2.0 * r * rp3 + rp * rm ) ;
57          p->dfdr[9] = t / 2.0 * sp * sm * ( -2.0 * r * rp5 + rp * rm ) ;
58          p->dfdr[10] = -t / 2.0 * sm * sp * ( (-rp5 * rp3 ) + rm * r2p8 ) ;
59          p->dfdr[11] = -t / 6.0 * sp * sm * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
60
61          p->dfds[3] = -t * 2.0 * rp3 * sp * rm * rp5 ;
62          p->dfds[0] = -t * 2.0 / 3.0 * rp3 * rp * sp * rp5 ;
63          p->dfds[1] = t / 3.0 * rp * rp3 * rp5 * ( s + 2.0 ) ;
```

```
64          p->dfds[2] = t * rm * rp3 * rp5 * ( s + 2.0 ) ;
65          p->dfds[4] = -t * rp * rm * rp5 * ( s + 2.0 ) ;
66          p->dfds[5] = t / 3.0 * rp * rm * rp3 * ( s + 2.0 ) ;
67          p->dfds[6] = -2.0 * t / 3.0 * rp * sp * rm * rp3 ;
68          p->dfds[7] = 2.0 * t * rp * rm * rp5 * sp ;
69          p->dfds[8] = t / 3.0 * rp * s * rm * rp3 ;
70          p->dfds[9] = -t * rp * s * rm * rp5 ;
71          p->dfds[10] = t * rp3 * s * rm * rp5 ;
72          p->dfds[11] = t / 3.0 * rp3 * s * rp * rp5 ;
73
74          return (0);
75  }
76  int     sh6_3x2b(x,p)
77  struct gausspts       *x;
78  struct shapefuncs     *p;
79
80  {
81          double r, s, t, rm, sm, rp, sp, rp3, rp5, r2p8, sp3 ;
82
83          r = x->y ;
84          s = x->x ;
85          if ( (r<-5.00001) || (r>1.00001) || (s<-3.00001) || (s>1.00001) )
86  return(12) ;
87
88          t = 1.0 / 64. ;
89          rm = 1.0 - r ;
90          rp = 1.0 + r ;
91          sp = 1.0 + s ;
92          sm = 1.0 - s ;
93          rp3 = 3.0 + r ;
94          rp5 = 5.0 + r ;
95          sp3 = 3.0 + s ;
96
97          p->dof = 12 ;
98
99          p->f[3] = t * rp3 * sm * rm * rp5 * sp3 ;
100         p->f[2] = t / 3.0 * rp3 * rp * sm * rp5 * sp3 ;
101         p->f[1] = t / 6.0 * rp * sp * rp3 * rp5 * sp3 ;
102         p->f[0] = t / 2.0 * rm * sp * rp3 * rp5 * sp3 ;
103         p->f[4] = -t / 2.0 * rp * sp * rm * rp5 * sp3 ;
104         p->f[5] = t / 6.0 * rp * sp * rm * rp3 * sp3;
105         p->f[6] = t / 3.0 * rp * sm * rm * rp3 * sp3;
106         p->f[7] = -t * rp * sm * rm * rp5 * sp3;
107         p->f[8] = -t / 6.0 * rp3 * sp * rp * rm * sm ;
108         p->f[9] = t / 2.0 * rp * sp * rm * rp5 * sm ;
109         p->f[10] = -t / 2.0 * rp3 * sp * rm * rp5 * sm ;
110         p->f[11] = -t / 6.0 * rp3 * sp * rp * rp5 * sm ;
111
112         if (x->w < 0.0)
113                 return(0) ;
114
115         r2p8 = 2.0 * r + 8.0 ;
116
117         p->dfds[3] = t * sm * sp3 * ( (-rp5 * rp3 ) + rm * r2p8 ) ;
118         p->dfds[2] = t / 3.0 * sm * sp3 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
119         p->dfds[1] = t / 6.0 * sp * sp3 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
120         p->dfds[0] = t / 2.0 * sp * sp3 * ( (-rp5 * rp3) + rm * r2p8 ) ;
121         p->dfds[4] = -t / 2.0 * sp * sp3 * ( -2.0 * r * rp5 + rp * rm ) ;
122         p->dfds[5] = t / 6.0 * sp * sp3 * ( -2.0 * r * rp3 + rp * rm ) ;
123         p->dfds[6] = t / 3.0 * sm * sp3 * ( -2.0 * r * rp3 + rp * rm ) ;
124         p->dfds[7] = -t * sm * sp3 * ( -2.0 * r * rp5 + rp * rm ) ;
125         p->dfds[8] = -t / 6.0 * sp * sm * ( -2.0 * r * rp3 + rp * rm ) ;
126         p->dfds[9] = t / 2.0 * sp * sm * ( -2.0 * r * rp5 + rp * rm ) ;
```

```
127         p->dfds[10] = -t / 2.0 * sm * sp * ( (-rp5 * rp3 ) + rm * r2p8 ) ;
128         p->dfds[11] = -t / 6.0 * sp * sm * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
129
130         p->dfdr[3] = -t * 2.0 * rp3 * sp * rm * rp5 ;
131         p->dfdr[2] = -t * 2.0 / 3.0 * rp3 * rp * sp * rp5 ;
132         p->dfdr[1] = t / 3.0 * rp * rp3 * rp5 * ( s + 2.0 ) ;
133         p->dfdr[0] = t * rm * rp3 * rp5 * ( s + 2.0 ) ;
134         p->dfdr[4] = -t * rp * rm * rp5 * ( s + 2.0 ) ;
135         p->dfdr[5] = t / 3.0 * rp * rm * rp3 * ( s + 2.0 ) ;
136         p->dfdr[6] = -2.0 * t / 3.0 * rp * sp * rm * rp3 ;
137         p->dfdr[7] = 2.0 * t * rp * rm * rp5 * sp ;
138         p->dfdr[8] = t / 3.0 * rp * s * rm * rp3 ;
139         p->dfdr[9] = -t * rp * s * rm * rp5 ;
140         p->dfdr[10] = t * rp3 * s * rm * rp5 ;
141         p->dfdr[11] = t / 3.0 * rp3 * s * rp * rp5 ;
142
143         return (0);
144 }
145
146 int     shcu5_3x3(x,p)
147 struct gausspts      *x;
148 struct shapefuncs    *p;
149
150 {
151         double r, s, t, rm, sm, rp, sp, rp3, rp5, r2p8, sp3, sp5, s2p8 ;
152
153         r = x->x ;
154         s = x->y ;
155         if ( (r<-5.00001) || (r>1.00001) || (s<-5.00001) || (s>1.00001) )
156 return(12) ;
157
158         t = 1.0 / 256. ;
159         rm = 1.0 - r ;
160         rp = 1.0 + r ;
161         sp = 1.0 + s ;
162         sm = 1.0 - s ;
163         rp3 = 3.0 + r ;
164         rp5 = 5.0 + r ;
165         sp3 = 3.0 + s ;
166         sp5 = 5.0 + s ;
167
168         p->dof = 12 ;
169
170         p->f[3] = t * rp3 * sm * rm * rp5 * sp3 * sp5 ;
171         p->f[0] = t / 3.0 * rp3 * rp * sm * rp5 * sp3 * sp5 ;
172         p->f[1] = t / 9.0 * rp * sp * rp3 * rp5 * sp3 * sp5 ;
173         p->f[2] = t / 3.0 * rm * sp * rp3 * rp5 * sp3 * sp5 ;
174         p->f[4] = -8.0 * t * rp * sp * rm * rp5 ;
175         p->f[5] = 8.0 * t / 3.0 * rp * sp * rm * rp3  ;
176         p->f[6] = 8.0 * t / 3.0 * rp * sm * rm * rp3  ;
177         p->f[7] = -8.0 * t * rp * sm * rm * rp5  ;
178         p->f[8] = -8.0 * t * sp * rm * sm * sp5 ;
179         p->f[11] = -8.0 * t * sp * rp * sm * sp5 ;
180         p->f[9] = 8.0 * t / 3.0 * sp * rm * sm * sp3 ;
181         p->f[10] = 8.0 * t / 3.0 * sp * rp * sm * sp3 ;
182
183         if (x->w < 0.0)
184               return(0) ;
185
186         r2p8 = 2.0 * r + 8.0 ;
187         s2p8 = 2.0 * s + 8.0 ;
188
189
```

```
190         p->dfdr[3] = t * sm * sp3 * sp5 * ( (-rp5 * rp3 ) + rm * r2p8 ) ;
191         p->dfdr[0] = t / 3.0 * sm * sp3 * sp5 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
192         p->dfdr[1] = t / 9.0 * sp * sp3 * sp5 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
193         p->dfdr[2] = t / 3.0 * sp * sp3 * sp5 * ( (-rp5 * rp3) + rm * r2p8 ) ;
194         p->dfdr[4] = -8.0 * t * sp * ( -2.0 * r * rp5 + rp * rm ) ;
195         p->dfdr[5] = 8.0 * t / 3.0 * sp * ( -2.0 * r * rp3 + rp * rm ) ;
196         p->dfdr[6] = 8.0 * t / 3.0 * sm * ( -2.0 * r * rp3 + rp * rm ) ;
197         p->dfdr[7] = -8.0 * t * sm * ( -2.0 * r * rp5 + rp * rm ) ;
198         p->dfdr[8] = 8.0 * t * sm * sp * sp5 ;
199         p->dfdr[11] = -8.0 * t * sp * sm * sp5 ;
200         p->dfdr[10] = 8.0 * t / 3.0 * sp * sm * sp3 ;
201         p->dfdr[9] = -8.0 * t / 3.0 * sm * sp * sp3 ;
202
203         p->dfds[3] = t * rp3 * rm * rp5 * ( -( sp5 * sp3 ) + sm * s2p8 ) ;
204         p->dfds[0] = t / 3.0 * rp3 * rp * rp5 * ( -( sp5 * sp3 ) + sm * s2p8 ) ;
205         p->dfds[1] = t / 9.0 * rp * rp3 * rp5 * ( ( sp5 * sp3 ) + sp * s2p8 ) ;
206         p->dfds[2] = t / 3.0 * rm * rp3 * rp5 * ( ( sp5 * sp3 ) + sp * s2p8 ) ;
207         p->dfds[4] = -8.0 * t * rp * rm * rp5 ;
208         p->dfds[5] = 8.0 * t / 3.0 * rp * rm * rp3 ;
209         p->dfds[6] = -8.0 * t / 3.0 * rp * rm * rp3 ;
210         p->dfds[7] = 8.0 * t * rp * rm * rp5 ;
211         p->dfds[8] = -8.0 * t * rm * ( -2.0 * s * sp5 + sp * sm ) ;
212         p->dfds[11] = -8.0 * t * rp * ( -2.0 * s * sp5 + sp * sm ) ;
213         p->dfds[9] = 8.0 * t / 3.0 * rm * ( -2.0 * s * sp3 + sp * sm ) ;
214         p->dfds[10] = 8.0 * t / 3.0 * rp * ( -2.0 * s * sp3 + sp * sm ) ;
215
216         return (0);
217 }
218
219 int     shcu6_3x3(x,p)
220 struct gausspts      *x;
221 struct shapefuncs    *p;
222
223 {
224         double r, s, t, rm, sm, rp, sp, rp3, rp5, r2p8, sp3, sp5, s2p8 ;
225
226         r = x->x ;
227         s = x->y ;
228         if ( (r<-5.00001) || (r>1.00001) || (s<-5.00001) || (s>1.00001) )
229 return(13) ;
230
231         t = 1.0 / 256. ;
232         rm = 1.0 - r ;
233         rp = 1.0 + r ;
234         sp = 1.0 + s ;
235         sm = 1.0 - s ;
236         rp3 = 3.0 + r ;
237         rp5 = 5.0 + r ;
238         sp3 = 3.0 + s ;
239         sp5 = 5.0 + s ;
240
241         p->dof = 13 ;
242
243         p->f[3] = t * rp3 * sm * rm * rp5 * sp3 * sp5 ;
244         p->f[0] = t / 3.0 * rp3 * rp * sm * rp5 * sp3 * sp5 ;
245         p->f[1] = t / 9.0 * rp * sp * rp3 * rp5 * sp3 * sp5 ;
246         p->f[2] = t / 3.0 * rm * sp * rp3 * rp5 * sp3 * sp5 ;
247         p->f[4] = -2.0 * t * rp * sp * rm * rp5 * sp3 ;
248         p->f[5] = 8.0 * t / 3.0 * rp * sp * rm * rp3 ;
249         p->f[6] = 8.0 * t / 3.0 * rp * sm * rm * rp3 ;
250         p->f[7] = -4.0 * t * rp * sm * rm * rp5 * sp3 ;
251         p->f[8] = 4.0 * t * rp * sp * rm * sm ;
252         p->f[9] = -4.0 * t * rp3 * sp * rm * sm * sp5 ;
```

```
253         p->f[10] = 8.0 * t / 3.0 * sp * rm * sm * sp3 ;
254         p->f[11] = 8.0 * t / 3.0 * sp * rp * sm * sp3 ;
255         p->f[12] = -2.0 * t * rp3 * sp * rp * sm * sp5 ;
256
257         if (x->w < 0.0)
258               return(0) ;
259
260         r2p8 = 2.0 * r + 8.0 ;
261         s2p8 = 2.0 * s + 8.0 ;
262
263
264         p->dfdr[3] = t * sm * sp3 * sp5 * ( (-rp5 * rp3 ) + rm * r2p8 ) ;
265         p->dfdr[0] = t / 3.0 * sm * sp3 * sp5 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
266         p->dfdr[1] = t / 9.0 * sp * sp3 * sp5 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
267         p->dfdr[2] = t / 3.0 * sp * sp3 * sp5 * ( (-rp5 * rp3) + rm * r2p8 ) ;
268         p->dfdr[4] = -2.0 * t * sp * sp3 * ( -2.0 * r * rp5 + rp * rm ) ;
269         p->dfdr[5] = 8.0 * t / 3.0 * sp * ( -2.0 * r * rp3 + rp * rm ) ;
270         p->dfdr[6] = 8.0 * t / 3.0 * sm * ( -2.0 * r * rp3 + rp * rm ) ;
271         p->dfdr[7] = -4.0 * t * sm * sp3 * ( -2.0 * r * rp5 + rp * rm ) ;
272         p->dfdr[8] = 4.0 * t * sm * sp * ( -2.0 * r ) ;
273         p->dfdr[9] = -4.0 * t * sm * sp * sp5 * (-2.0 * rp ) ;
274         p->dfdr[10] = -8.0 * t / 3.0 * sm * sp * sp3 ;
275         p->dfdr[11] = 8.0 * t / 3.0 * sp * sm * sp3  ;
276         p->dfdr[12] = -4.0 * t * sp * sm * sp5 * ( r + 2.0 ) ;
277
278         p->dfds[3] = t * rp3 * rm * rp5 * ( -( sp5 * sp3 ) + sm * s2p8 ) ;
279         p->dfds[0] = t / 3.0 * rp3 * rp * rp5 * ( -( sp5 * sp3 ) + sm * s2p8 ) ;
280         p->dfds[1] = t / 9.0 * rp * rp3 * rp5 * ( ( sp5 * sp3 ) + sp * s2p8 ) ;
281         p->dfds[2] = t / 3.0 * rm * rp3 * rp5 * ( ( sp5 * sp3 ) + sp * s2p8 ) ;
282         p->dfds[4] = -2.0 * t * rp * rm * rp5 * ( 2.0 * s + 4.0 ) ;
283         p->dfds[5] = 8.0 * t / 3.0 * rp * rm * rp3 ;
284         p->dfds[6] = -8.0 * t / 3.0 * rp * rm * rp3 ;
285         p->dfds[7] = -4.0 * t * rp * rm * rp5 * ( -2.0 * sp ) ;
286         p->dfds[8] = 4.0 * t * rp * rm * ( -2.0 * s ) ;
287         p->dfds[9] = -4.0 * t * rp3 * rm * ( -2.0 * s * sp5 + sp * sm ) ;
288         p->dfds[10] = 8.0 * t / 3.0 * rm * ( -2.0 * s * sp3 + sp * sm ) ;
289         p->dfds[11] = 8.0 * t / 3.0 * rp * ( -2.0 * s * sp3 + sp * sm ) ;
290         p->dfds[12] = -2.0 * t * rp3 * rp * ( -2.0 * s * sp5 + sp * sm ) ;
291
292         return (0);
293   }
294
```

```
1   /*    . the shape functions for the cubic upwinding functions   239          */
2   #include "fe.h"
3
4
5   extern struct control  N ;
6
7   int     shcu7_3x3(x,p)
8   struct gausspts      *x;
9   struct shapefuncs    *p;
10
11  {
12          double r, s, t, rm, sm, rp, sp, rp3, rp5, r2p8, sp3, sp5, s2p8 ;
13
14          r = x->x ;
15          s = x->y ;
16          if ( (r<-5.00001) || (r>1.00001) || (s<-5.00001) || (s>1.00001) )
17  return(14) ;
18
19          t = 1.0 / 256. ;
20          rm = 1.0 - r ;
21          rp = 1.0 + r ;
22          sp = 1.0 + s ;
23          sm = 1.0 - s ;
24          rp3 = 3.0 + r ;
25          rp5 = 5.0 + r ;
26          sp3 = 3.0 + s ;
27          sp5 = 5.0 + s ;
28
29          p->dof = 14 ;
30
31          p->f[3] = t * rp3 * sm * rm * rp5 * sp3 * sp5 ;
32          p->f[0] = t / 3.0 * rp3 * rp * sm * rp5 * sp3 * sp5 ;
33          p->f[1] = t / 9.0 * rp * sp * rp3 * rp5 * sp3 * sp5 ;
34          p->f[2] = t / 3.0 * rm * sp * rp3 * rp5 * sp3 * sp5 ;
35          p->f[4] = -2.0 * t * rp * sp * rm * rp5 * sp3 ;
36          p->f[5] = 2.0 * t / 3.0 * rp * sp * sp3 * rm * rp3 ;
37          p->f[6] = 4.0 * t / 3.0 * rp * sm * sp3 * rm * rp3 ;
38          p->f[7] = -4.0 * t * rp * sm * rm * rp5 * sp3 ;
39          p->f[8] = -2.0 / 3.0 * t * rp * sp * rm * sm * rp3 ;
40          p->f[9] = 2.0 * t * rp * sp * rm * sm * rp5 ;
41          p->f[10] = -t * rp3 * sp * rm * rp5 * sm * sp5 ;
42          p->f[11] = 8.0 * t / 3.0 * sp * rm * sm * sp3 ;
43          p->f[12] = 8.0 * t / 3.0 * sp * rp * sm * sp3 ;
44          p->f[13] = -t / 3.0 * rp3 * rp5 * sp * rp * sm * sp5 ;
45
46          if (x->w < 0.0)
47                  return(0) ;
48
49          r2p8 = 2.0 * r + 8.0 ;
50          s2p8 = 2.0 * s + 8.0 ;
51
52          p->dfdr[3] = t * sm * sp3 * sp5 * ( (-rp5 * rp3 ) + rm * r2p8 ) ;
53          p->dfdr[0] = t / 3.0 * sm * sp3 * sp5 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
54          p->dfdr[1] = t / 9.0 * sp * sp3 * sp5 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
55          p->dfdr[2] = t / 3.0 * sp * sp3 * sp5 * ( (-rp5 * rp3) + rm * r2p8 ) ;
56          p->dfdr[4] = -2.0 * t * sp * sp3 * ( -2.0 * r * rp5 + rp * rm ) ;
57          p->dfdr[5] = 2.0 * t / 3.0 * sp * sp3 * ( -2.0 * r * rp3 + rp * rm ) ;
58          p->dfdr[6] = 4.0 * t / 3.0 * sm * sp3 * ( -2.0 * r * rp3 + rp * rm ) ;
59          p->dfdr[7] = -4.0 * t * sm * sp3 * ( -2.0 * r * rp5 + rp * rm ) ;
60          p->dfdr[8] = -2.0 / 3.0 * t * sp * sm * ( -2.0 * r * rp3 + rp * rm ) ;
61          p->dfdr[9] = 2.0 * t * sm * sp * ( -2.0 * r * rp5 + rp * rm ) ;
62          p->dfdr[10] = -t * sm * sp * sp5 * ( (-rp5 * rp3 ) + rm * r2p8 ) ;
63          p->dfdr[11] = -8.0 * t / 3.0 * sm * sp * sp3 ;
```

```
64          p->dfdr[12] = 8.0 * t / 3.0 * sp * sm * sp3  ;
65          p->dfdr[13] = -t / 3.0 * sp * sm * sp5 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
66
67          p->dfds[3] = t * rp3 * rm * rp5 * ( -( sp5 * sp3 ) + sm * s2p8 ) ;
68          p->dfds[0] = t / 3.0 * rp3 * rp * rp5 * ( -( sp5 * sp3 ) + sm * s2p8 ) ;
69          p->dfds[1] = t / 9.0 * rp * rp3 * rp5 * ( ( sp5 * sp3 ) + sp * s2p8 ) ;
70          p->dfds[2] = t / 3.0 * rm * rp3 * rp5 * ( ( sp5 * sp3 ) + sp * s2p8 ) ;
71          p->dfds[4] = -2.0 * t * rp * rm * rp5 * ( 2.0 * s + 4.0 ) ;
72          p->dfds[5] = 2.0 * t / 3.0 * rp * rm * rp3 * ( 2.0 * s + 4.0 ) ;
73          p->dfds[6] = 4.0 * t / 3.0 * rp * rm * rp3 * (-2.0 * sp );
74          p->dfds[7] = -4.0 * t * rp * rm * rp5 * ( -2.0 * sp ) ;
75          p->dfds[8] = -2.0 / 3.0 * t * rp * rm * rp3 * ( -2.0 * s) ;
76          p->dfds[9] = 2.0 * t * rp * rm * rp5 * ( -2.0 * s ) ;
77          p->dfds[10] = -t * rp3 * rm * rp5 * ( -2.0 * s * sp5 + sp * sm ) ;
78          p->dfds[11] = 8.0 * t / 3.0 * rm * ( -2.0 * s * sp3 + sp * sm ) ;
79          p->dfds[12] = 8.0 * t / 3.0 * rp * ( -2.0 * s * sp3 + sp * sm ) ;
80          p->dfds[13] = -t / 3.0 * rp3 * rp * rp5 * ( -2.0 * s * sp5 + sp * sm ) ;
81
82          return (0);
83  }
84
85  int     sh7_3x3b(x,p)
86  struct gausspts       *x;
87  struct shapefuncs     *p;
88
89  {
90          double r, s, t, rm, sm, rp, sp, rp3, rp5, r2p8, sp3, sp5, s2p8 ;
91
92          r = x->y ;
93          s = x->x ;
94          if ( (r<-5.00001) || (r>1.00001) || (s<-5.00001) || (s>1.00001) )
95  return(14) ;
96
97          t = 1.0 / 256. ;
98          rm = 1.0 - r ;
99          rp = 1.0 + r ;
100         sp = 1.0 + s ;
101         sm = 1.0 - s ;
102         rp3 = 3.0 + r ;
103         rp5 = 5.0 + r ;
104         sp3 = 3.0 + s ;
105         sp5 = 5.0 + s ;
106
107         p->dof = 14 ;
108
109         p->f[3] = t * rp3 * sm * rm * rp5 * sp3 * sp5 ;
110         p->f[2] = t / 3.0 * rp3 * rp * sm * rp5 * sp3 * sp5 ;
111         p->f[1] = t / 9.0 * rp * sp * rp3 * rp5 * sp3 * sp5 ;
112         p->f[0] = t / 3.0 * rm * sp * rp3 * rp5 * sp3 * sp5 ;
113         p->f[4] = -2.0 * t * rp * sp * rm * rp5 * sp3 ;
114         p->f[5] = 2.0 * t / 3.0 * rp * sp * sp3 * rm * rp3 ;
115         p->f[6] = 4.0 * t / 3.0 * rp * sm * sp3 * rm * rp3 ;
116         p->f[7] = -4.0 * t * rp * sm * rm * rp5 * sp3 ;
117         p->f[8] = -2.0 / 3.0 * t * rp * sp * rm * sm * rp3 ;
118         p->f[9] = 2.0 * t * rp * sp * rm * sm * rp5 ;
119         p->f[10] = -t * rp3 * sp * rm * rp5 * sm * sp5 ;
120         p->f[11] = 8.0 * t / 3.0 * sp * rm * sm * sp3 ;
121         p->f[12] = 8.0 * t / 3.0 * sp * rp * sm * sp3 ;
122         p->f[13] = -t / 3.0 * rp3 * rp5 * sp * rp * sm * sp5 ;
123
124         if (x->w < 0.0)
125                 return(0) ;
126
```

```
127         r2p8 = 2.0 * r + 8.0 ;
128         s2p8 = 2.0 * s + 8.0 ;
129
130         p->dfds[3] = t * sm * sp3 * sp5 * ( (-rp5 * rp3 ) + rm * r2p8 ) ;
131         p->dfds[2] = t / 3.0 * sm * sp3 * sp5 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
132         p->dfds[1] = t / 9.0 * sp * sp3 * sp5 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
133         p->dfds[0] = t / 3.0 * sp * sp3 * sp5 * ( (-rp5 * rp3) + rm * r2p8 ) ;
134         p->dfds[4] = -2.0 * t * sp * sp3 * ( -2.0 * r * rp5 + rp * rm ) ;
135         p->dfds[5] = 2.0 * t / 3.0 * sp * sp3 * ( -2.0 * r * rp3 + rp * rm ) ;
136         p->dfds[6] = 4.0 * t / 3.0 * sm * sp3 * ( -2.0 * r * rp3 + rp * rm ) ;
137         p->dfds[7] = -4.0 * t * sm * sp3 * ( -2.0 * r * rp5 + rp * rm ) ;
138         p->dfds[8] = -2.0 / 3.0 * t * sp * sm * ( -2.0 * r * rp3 + rp * rm ) ;
139         p->dfds[9] = 2.0 * t * sm * sp * ( -2.0 * r * rp5 + rp * rm ) ;
140         p->dfds[10] = -t * sm * sp * sp5 * ( (-rp5 * rp3 ) + rm * r2p8 ) ;
141         p->dfds[11] = -8.0 * t / 3.0 * sm * sp * sp3 ;
142         p->dfds[12] = 8.0 * t / 3.0 * sp * sm * sp3  ;
143         p->dfds[13] = -t / 3.0 * sp * sm * sp5 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
144
145         p->dfdr[3] = t * rp3 * rm * rp5 * ( -( sp5 * sp3 ) + sm * s2p8 ) ;
146         p->dfdr[2] = t / 3.0 * rp3 * rp * rp5 * ( -( sp5 * sp3 ) + sm * s2p8 ) ;
147         p->dfdr[1] = t / 9.0 * rp * rp3 * rp5 * ( ( sp5 * sp3 ) + sp * s2p8 ) ;
148         p->dfdr[0] = t / 3.0 * rm * rp3 * rp5 * ( ( sp5 * sp3 ) + sp * s2p8 ) ;
149         p->dfdr[4] = -2.0 * t * rp * rm * rp5 * ( 2.0 * s + 4.0 ) ;
150         p->dfdr[5] = 2.0 * t / 3.0 * rp * rm * rp3 * ( 2.0 * s + 4.0 ) ;
151         p->dfdr[6] = 4.0 * t / 3.0 * rp * rm * rp3 * (-2.0 * sp );
152         p->dfdr[7] = -4.0 * t * rp * rm * rp5 * ( -2.0 * sp ) ;
153         p->dfdr[8] = -2.0 / 3.0 * t * rp * rm * rp3 * ( -2.0 * s) ;
154         p->dfdr[9] = 2.0 * t * rp * rm * rp5 * ( -2.0 * s ) ;
155         p->dfdr[10] = -t * rp3 * rm * rp5 * ( -2.0 * s * sp5 + sp * sm ) ;
156         p->dfdr[11] = 8.0 * t / 3.0 * rm * ( -2.0 * s * sp3 + sp * sm ) ;
157         p->dfdr[12] = 8.0 * t / 3.0 * rp * ( -2.0 * s * sp3 + sp * sm ) ;
158         p->dfdr[13] = -t / 3.0 * rp3 * rp * rp5 * ( -2.0 * s * sp5 + sp * sm ) ;
159
160         return (0);
161 }
162
163 int     shcu8_3x3(x,p)
164 struct gausspts       *x;
165 struct shapefuncs     *p;
166
167 {
168         double r, s, t, rm, sm, rp, sp, rp3, rp5, r2p8, sp3, sp5, s2p8 ;
169
170         r = x->x ;
171         s = x->y ;
172         if ( (r<-5.00001) || (r>1.00001) || (s<-5.00001) || (s>1.00001) )
173 return(15) ;
174
175         t = 1.0 / 256. ;
176         rm = 1.0 - r ;
177         rp = 1.0 + r ;
178         sp = 1.0 + s ;
179         sm = 1.0 - s ;
180         rp3 = 3.0 + r ;
181         rp5 = 5.0 + r ;
182         sp3 = 3.0 + s ;
183         sp5 = 5.0 + s ;
184
185         p->dof = 15 ;
186
187
188
189         p->f[3] = t * rp3 * sm * rm * rp5 * sp3 * sp5 ;
```

```
190         p->f[0] = t / 3.0 * rp3 * rp * sm * rp5 * sp3 * sp5 ;
191         p->f[1] = t / 9.0 * rp * sp * rp3 * rp5 * sp3 * sp5 ;
192         p->f[2] = t / 3.0 * rm * sp * rp3 * rp5 * sp3 * sp5 ;
193         p->f[4] = -t / 3.0 * rp * sp * rm * rp5 * sp3 * sp5 ;
194         p->f[5] = 2.0 * t / 3.0 * rp * sp * sp3 * rm * rp3 ;
195         p->f[6] = 4.0 * t / 3.0 * rp * sm * sp3 * rm * rp3 ;
196         p->f[7] = -t * rp * sm * rm * rp5 * sp3 * sp5 ;
197         p->f[8] = -2.0 / 3.0 * t * rp * sp * rm * sm * rp3 ;
198         p->f[9] = t * rp * sp * rm * sm * sp5 * rp5 ;
199         p->f[10] = -t * rp3 * sp * rm * rp5 * sm * sp5 ;
200         p->f[11] = -2.0 / 3.0 * t * sp * rm * rp * sm * sp3 ;
201         p->f[12] = 4.0 * t / 3.0 * sp * rp3 * rm * sm * sp3 ;
202         p->f[13] = 2.0 * t / 3.0 * sp * rp3 * rp * sm * sp3 ;
203         p->f[14] = -t / 3.0 * rp3 * rp5 * sp * rp * sm * sp5 ;
204
205         if (x->w < 0.0)
206               return(0) ;
207         r2p8 = 2.0 * r + 8.0 ;
208         s2p8 = 2.0 * s + 8.0 ;
209
210         p->dfdr[3] = t * sm * sp3 * sp5 * ( (-rp5 * rp3 ) + rm * r2p8 ) ;
211         p->dfdr[0] = t / 3.0 * sm * sp3 * sp5 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
212         p->dfdr[1] = t / 9.0 * sp * sp3 * sp5 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
213         p->dfdr[2] = t / 3.0 * sp * sp3 * sp5 * ( (-rp5 * rp3) + rm * r2p8 ) ;
214         p->dfdr[4] = -t / 3.0 * sp5 * sp * sp3 * ( -2.0 * r * rp5 + rp * rm ) ;
215         p->dfdr[5] = 2.0 * t / 3.0 * sp3 * sp * ( -2.0 * r * rp3 + rp * rm ) ;
216         p->dfdr[6] = 4.0 * t / 3.0 * sp3 * sm * ( -2.0 * r * rp3 + rp * rm ) ;
217         p->dfdr[7] = -t * sm * sp3 * sp5 * ( -2.0 * r * rp5 + rp * rm ) ;
218         p->dfdr[8] = -2.0 / 3.0 * t * sp * sm * ( -2.0 * r * rp3 + rp * rm ) ;
219         p->dfdr[9] = t * sm * sp * sp5 * ( -2.0 * r * rp5 + rp * rm ) ;
220         p->dfdr[10] = -t * sm * sp * sp5 * ( (-rp5 * rp3 ) + rm * r2p8 ) ;
221         p->dfdr[11] = -2.0 / 3.0 * t * sm * sp * sp3 * ( -2.0 * r ) ;
222         p->dfdr[12] = -2.0 * t / 3.0 * sm * sp * sp3 * rp ;
223         p->dfdr[13] = 2.0 * t / 3.0 * sp * sm * sp3 * ( 2.0 * r + 4.0 ) ;
224         p->dfdr[14] = -t / 3.0 * sp * sm * sp5 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
225
226         p->dfds[3] = t * rp3 * rm * rp5 * ( -( sp5 * sp3 ) + sm * s2p8 ) ;
227         p->dfds[0] = t / 3.0 * rp3 * rp * rp5 * ( -( sp5 * sp3 ) + sm * s2p8 ) ;
228         p->dfds[1] = t / 9.0 * rp * rp3 * rp5 * ( ( sp5 * sp3 ) + sp * s2p8 ) ;
229         p->dfds[2] = t / 3.0 * rm * rp3 * rp5 * ( ( sp5 * sp3 ) + sp * s2p8 ) ;
230         p->dfds[4] = -t / 3.0 * rp * rm * rp5 * ( ( sp5 * sp3 ) + sp * s2p8 ) ;
231         p->dfds[5] = 2.0 * t / 3.0 * rp * rm * rp3 * ( 2.0 * s + 4.0 ) ;
232         p->dfds[6] = 4.0 * t / 3.0 * rp * rm * rp3 * ( -2.0 * sp ) ;
233         p->dfds[7] = -t * rp * rm * rp5 * ( -( sp5 * sp3 ) + sm * s2p8 ) ;
234         p->dfds[8] = -2.0 / 3.0 * t * rp * rm * rp3 * ( -2.0 * s) ;
235         p->dfds[9] = t * rp * rm * rp5 * ( -2.0 * s * sp5 + sp * sm ) ;
236         p->dfds[10] = -t * rp3 * rm * rp5 * ( -2.0 * s * sp5 + sp * sm ) ;
237         p->dfds[11] = -2.0 / 3.0 * t * rm * rp * ( -2.0 * s * sp3 + sp * sm ) ;
238         p->dfds[12] = 4.0 * t / 3.0 * rm * rp3 * ( -2.0 * s * sp3 + sp * sm ) ;
239         p->dfds[13] = 2.0 * t / 3.0 * rp * rp3 * ( -2.0 * s * sp3 + sp * sm ) ;
240         p->dfds[14] = -t / 3.0 * rp3 * rp * rp5 * ( -2.0 * s * sp5 + sp * sm ) ;
241
242         return (0);
243   }
244
```

```
1   /*      the shape functions for the cubic upwinding functions   239          */
2   #include "fe.h"
3
4
5   extern struct control  N ;
6
7   int     shcu9_3x3(x,p)
8   struct gausspts       *x;
9   struct shapefuncs     *p;
10
11  {
12          double r, s, t, rm, sm, rp, sp, rp3, rp5, r2p8, sp3, sp5, s2p8 ;
13
14          r = x->x ;
15          s = x->y ;
16          if ( (r<-5.00001) || (r>1.00001) || (s<-5.00001) || (s>1.00001) )
17  return(16) ;
18
19          t = 1.0 / 256. ;
20          rm = 1.0 - r ;
21          rp = 1.0 + r ;
22          sp = 1.0 + s ;
23          sm = 1.0 - s ;
24          rp3 = 3.0 + r ;
25          rp5 = 5.0 + r ;
26          sp3 = 3.0 + s ;
27          sp5 = 5.0 + s ;
28
29          p->dof = 16 ;
30
31          p->f[3] = t * rp3 * sm * rm * rp5 * sp3 * sp5 ;
32          p->f[0] = t / 3.0 * rp3 * rp * sm * rp5 * sp3 * sp5 ;
33          p->f[1] = t / 9.0 * rp * sp * rp3 * rp5 * sp3 * sp5 ;
34          p->f[2] = t / 3.0 * rm * sp * rp3 * rp5 * sp3 * sp5 ;
35          p->f[4] = -t / 3.0 * rp * sp * rm * rp5 * sp3 * sp5 ;
36          p->f[5] = t / 9.0 * rp * sp * rm * rp3 * sp3 * sp5 ;
37          p->f[6] = t / 3.0 * rp * sm * rm * rp3 * sp3 * sp5 ;
38          p->f[7] = -t * rp * sm * rm * rp5 * sp3 * sp5 ;
39          p->f[8] = -t / 3.0 * rp * sm * rm * rp3 * sp * sp5 ;
40          p->f[9] = t * rp * sp * rm * rp5 * sm * sp5 ;
41          p->f[10] = -t * rp3 * sp * rm * rp5 * sm * sp5 ;
42          p->f[11] = t / 9.0 * rp3 * sp * rp * rm * sm * sp3 ;
43          p->f[12] = -t / 3.0 * rp * sp * rm * rp5 * sm * sp3 ;
44          p->f[13] = t / 3.0 * rp3 * sp * rm * rp5 * sm * sp3 ;
45          p->f[14] = t / 9.0 * rp3 * sp * rp * rp5 * sm * sp3 ;
46          p->f[15] = -t / 3.0 * rp3 * sp * rp * rp5 * sm * sp5 ;
47
48          if (x->w < 0.0)
49                  return(0) ;
50
51          r2p8 = 2.0 * r + 8.0 ;
52          s2p8 = 2.0 * s + 8.0 ;
53
54
55          p->dfdr[3] = t * sm * sp3 * sp5 * ( (-rp5 * rp3 ) + rm * r2p8 ) ;
56          p->dfdr[0] = t / 3.0 * sm * sp3 * sp5 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
57          p->dfdr[1] = t / 9.0 * sp * sp3 * sp5 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
58          p->dfdr[2] = t / 3.0 * sp * sp3 * sp5 * ( (-rp5 * rp3) + rm * r2p8 ) ;
59          p->dfdr[4] = -t / 3.0 * sp * sp3 * sp5 * ( -2.0 * r * rp5 + rp * rm ) ;
60          p->dfdr[5] = t / 9.0 * sp * sp3 * sp5 * ( -2.0 * r * rp3 + rp * rm ) ;
61          p->dfdr[6] = t / 3.0 * sm * sp3 * sp5 * ( -2.0 * r * rp3 + rp * rm ) ;
62          p->dfdr[7] = -t * sm * sp3 * sp5 * ( -2.0 * r * rp5 + rp * rm ) ;
63          p->dfdr[8] = -t / 3.0 * sm * sp * sp5 * ( -2.0 * r * rp3 + rp * rm ) ;
```

```
64          p->dfdr[9] = t * sm * sp5 * sp * ( -2.0 * r * rp5 + rp * rm ) ;
65          p->dfdr[10] = -t * sm * sp * sp5 * ( (-rp5 * rp3 ) + rm * r2p8 ) ;
66          p->dfdr[11] = t / 9.0 * sm * sp * sp3 * ( -2.0 * r * rp3 + rp * rm ) ;
67          p->dfdr[12] = -t / 3.0 * sm * sp3 * sp * ( -2.0 * r * rp5 + rp * rm ) ;
68          p->dfdr[13] = t / 3.0 * sm * sp * sp3 * ( (-rp5 * rp3 ) + rm * r2p8 ) ;
69          p->dfdr[14] = t / 9.0 * sp * sm * sp3 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
70          p->dfdr[15] = -t / 3.0 * sp * sm * sp5 * ( ( rp5 * rp3 ) + rp * r2p8 ) ;
71
72          p->dfds[3] = t * rp3 * rm * rp5 * ( -( sp5 * sp3 ) + sm * s2p8 ) ;
73          p->dfds[0] = t / 3.0 * rp3 * rp * rp5 * ( -( sp5 * sp3 ) + sm * s2p8 ) ;
74          p->dfds[1] = t / 9.0 * rp * rp3 * rp5 * ( -( sp5 * sp3 ) + sp * s2p8 ) ;
75          p->dfds[2] = t / 3.0 * rm * rp3 * rp5 * ( ( sp5 * sp3 ) + sp * s2p8 ) ;
76          p->dfds[4] = -t / 3.0 * rp * rm * rp5 * ( ( sp5 * sp3 ) + sp * s2p8 ) ;
77          p->dfds[5] = t / 9.0 * rp * rm * rp3  * ( ( sp5 * sp3 ) + sp * s2p8 ) ;
78          p->dfds[6] = t / 3.0 * rp * rm * rp3  * ( -( sp5 * sp3 ) + sm * s2p8 ) ;
79          p->dfds[7] = -t * rp * rm * rp5  * ( -( sp5 * sp3 ) + sm * s2p8 ) ;
80          p->dfds[8] = -t / 3.0 * rp * rm * rp3 * ( -2.0 * s * sp5 + sp * sm ) ;
81          p->dfds[9] = t * rp * rm * rp5 * ( -2.0 * s * sp5 + sp * sm ) ;
82          p->dfds[10] = -t * rp3 * rm * rp5 * ( -2.0 * s * sp5 + sp * sm ) ;
83          p->dfds[11] = t / 9.0 * rp3 * rp * rm * ( -2.0 * s * sp3 + sp * sm ) ;
84          p->dfds[12] = -t / 3.0 * rp * rm * rp5 * ( -2.0 * s * sp3 + sp * sm ) ;
85          p->dfds[13] = t / 3.0 * rp3 * rm * rp5 * ( -2.0 * s * sp3 + sp * sm ) ;
86          p->dfds[14] = t / 9.0 * rp3 * rp * rp5 * ( -2.0 * s * sp3 + sp * sm ) ;
87          p->dfds[15] = -t / 3.0 * rp3 * rp * rp5 * ( -2.0 * s * sp5 + sp * sm ) ;
88
89          return (0);
90
91  }
```

Example Input File for a River Mesh Generation

1	40	5	20	20			
2	x	z	h	u	v	Elong	Elat
3	212.9	459.9	0	0	0	0.000	0.000
4	213	460	0.06	0.064	0	0.004	0.002
5	215.55	461.75	0.31	0.356	0	0.020	0.014
6	218.1	463.5	0.51	0.456	0	0.026	0.017
7	220.65	465.25	0.55	0.481	0	0.027	0.018
8	223.2	467	0.48	0.436	0	0.025	0.017
9	225.75	468.75	0.6	0.436	0	0.025	0.017
10	228.3	471.1	0.66	0.483	0	0.028	0.018
11	230.85	472.25	0.73	0.512	0	0.029	0.020
12	233.4	474	0.58	0.504	0	0.029	0.019
13	235.95	475.75	0.63	0.49	0	0.028	0.019
14	238.5	477.5	0.48	0.523	0	0.030	0.020
15	241.05	479.25	0.54	0.473	0	0.027	0.018
16	243.6	481	0.53	0.511	0	0.029	0.019
17	246.15	482.75	0.42	0.488	0	0.028	0.019
18	248.7	484.5	0.4	0.439	0	0.025	0.017
19	251.25	486.25	0.74	0.333	0	0.019	0.013
20	253.8	488	0.15	0.178	0	0.010	0.007
21	256.35	488.8	0	0	0	0.000	0.000
22							
23	258.9	491.5	0	0	0	0.000	0.000
24	261.45	493.25	0	0	0	0.000	0.000
25	x	z	h	u	v	Elong	Elat
26	463	360	0	0	0	0.000	0.000
27	463.6	362.75	0.3	0.184	0	0.013	0.009
28	464.2	365.5	0.56	0.25	0	0.017	0.012
29	464.8	368.25	0.5	0.205	0	0.014	0.010
30	465.4	371	0.6	0.273	0	0.019	0.013
31	466	373.75	0.71	0.194	0	0.013	0.009
32	466.6	376.5	0.7	0.328	0	0.023	0.015
33	467.2	379.25	0.7	0.31	0	0.022	0.014
34	467.8	382	0.74	0.337	0	0.023	0.016
35	468.4	384.75	0.8	0.284	0	0.020	0.013
36	469	387.5	0.74	0.333	0	0.023	0.015
37	469.6	390.25	0.7	0.339	0	0.024	0.016
38	470.2	393	0.62	0.334	0	0.023	0.015
39	470.8	395.75	0.7	0.333	0	0.023	0.015
40	471.4	398.5	0.61	0.354	0	0.025	0.016
41	472	401.25	0.6	0.328	0	0.023	0.015
42	472.6	404	0.575	0.298	0	0.021	0.014
43	473.2	406.75	0.51	0.302	0	0.021	0.014
44	473.8	409.5	0.415	0.247	0	0.017	0.011
45	474.4	412.25	0.2	0.184	0	0.013	0.009
46	475	415	0	0	0	0.000	0.000
47	x	z	h	u	v	Elong	Elat
48	713	320	0	0	0	0.000	0.000
49	713	323	0.24	0.16	0	0.011	0.007
50	713	326	0.25	0.233	0	0.016	0.011
51	713	329	0.36	0.264	0	0.018	0.012
52	713	332	0.43	0.306	0	0.021	0.014
53	713	335	0.47	0.434	0	0.030	0.020
54	713	338	0.42	0.413	0	0.029	0.019
55	713	341	0.48	0.384	0	0.027	0.018
56	713	344	0.48	0.397	0	0.028	0.018
57	713	347	0.5	0.464	0	0.032	0.022
58	713	350	0.44	0.462	0	0.032	0.021
59	713	353	0.47	0.518	0	0.036	0.024
60	713	356	0.47	0.462	0	0.032	0.021
61	713	359	0.51	0.514	0	0.036	0.024
62	713	362	0.5	0.491	0	0.034	0.023
63	713	365	0.51	0.514	0	0.036	0.024

64	713	368	0.52	0.518	0	0.036	0.024
65	713	371	0.49	0.525	0	0.037	0.024
66	713	374	0.46	0.464	0	0.032	0.022
67	713	377	0.3	0.356	0	0.025	0.017
68	713	380	0	0	0	0.000	0.000
69	x	z	h	u	v	Elong	Elat
70	963	317	0	0	0	0.000	0.000
71	963	320	0.09	0.107	0	0.007	0.005
72	963	323	0.33	0.72	0	0.046	0.031
73	963	326	0.4	0.904	0	0.058	0.039
74	963	329	0.5	0.965	0	0.062	0.042
75	963	332	0.53	0.956	0	0.062	0.041
76	963	335	0.46	0.876	0	0.057	0.038
77	963	338	0.38	0.802	0	0.052	0.035
78	963	341	0.29	0.796	0	0.051	0.034
79	963	344	0.23	0.519	0	0.034	0.022
80	963	347	0.17	0.634	0	0.041	0.027
81	963	350	0.21	0.605	0	0.039	0.026
82	963	353	0.26	0.619	0	0.040	0.027
83	963	356	0.2	0.365	0	0.024	0.016
84	963	359	0.13	0.307	0	0.020	0.013
85	963	362	0.11	0.551	0	0.036	0.024
86	963	365	0.19	0.495	0	0.032	0.021
87	963	368	0.29	0.507	0	0.033	0.022
88	963	371	0.3	0.657	0	0.042	0.028
89	963	374	0.29	0.313	0	0.020	0.013
90	963	377	0	0	0	0.000	0.000
91	x	z	h	u	v	Elong	Elat
92	1213	220	0	0	0	0.000	0.000
93	1214	222.5	0.21	0.144	0	0.010	0.006
94	1215	225	0.35	0.225	0	0.015	0.010
95	1216	227.5	0.46	0.224	0	0.015	0.010
96	1217	230	0.49	0.252	0	0.017	0.011
97	1218	232.5	0.55	0.31	0	0.020	0.014
98	1219	235	0.63	0.334	0	0.022	0.015
99	1220	237.5	0.74	0.326	0	0.022	0.014
100	1221	240	0.79	0.315	0	0.021	0.014
101	1222	242.5	0.84	0.339	0	0.022	0.015
102	1223	245	0.92	0.339	0	0.022	0.015
103	1224	247.5	1.03	0.328	0	0.022	0.014
104	1225	250	1	0.311	0	0.021	0.014
105	1226	252.5	1	0.31	0	0.020	0.014
106	1227	255	1	0.246	0	0.016	0.011
107	1228	257.5	0.83	0.256	0	0.017	0.011
108	1229	260	0.64	0.233	0	0.015	0.010
109	1230	262.5	0.6	0.114	0	0.008	0.005
110	1231	265	0.4	0.039	0	0.003	0.002
111	1232	267.5	0	0	0	0.000	0.000
112	1233	270	0	0	0	0.000	0.000
113	x	z	h	u	v	Elong	Elat
114	1463	0	0	0	0	0.000	0.000
115	1464.9	2.5	0.3	0.129	0	0.012	0.008
116	1466.8	5	0.41	0.145	0	0.013	0.009
117	1468.7	7.5	0.64	0.046	0	0.004	0.003
118	1470.6	10	0.68	0.031	0	0.003	0.002
119	1472.5	12.5	0.7	0.038	0	0.004	0.002
120	1474.4	15	0.92	0.038	0	0.004	0.002
121	1476.3	17.5	0.92	0.269	0	0.025	0.017
122	1478.2	20	0.74	0.31	0	0.029	0.019
123	1480.1	22.5	0.57	0.334	0	0.031	0.021
124	1482	25	0.52	0.391	0	0.036	0.024
125	1483.9	27.5	0.55	0.36	0	0.033	0.022
126	1485.8	30	0.57	0.412	0	0.038	0.025

127	1487.7	32.5	0.58	0.422	0	0.039	0.026
128	1489.6	35	0.58	0.397	0	0.037	0.024
129	1491.5	37.5	0.61	0.359	0	0.033	0.022
130	1493.4	40	0.59	0.368	0	0.034	0.023
131	1495.3	42.5	0.6	0.294	0	0.027	0.018
132	1497.2	45	0.58	0.344	0	0.032	0.021
133	1499.1	47.5	0.56	0.284	0	0.026	0.017
134	1501	50	0.57	0.235	0	0.022	0.014

Example Output of a Mesh

Transient analysis = 1
Number of Time Steps = 10
Delta t Acceleration Factor = 1.000
Time = 0.00000
Delta t = 10.00000
Theta = 0.500
Dimensions = 2
Number of Variables = 1
[K] governing equation numbers
1 1
Number of Parameters = 8
Number of Boundary Parameters = -2
Number of Nodes = 231
Number of Elements = 200
Number of Boundary Elements = 60
Number of Element in X direction = 10
Number of Blocks in X direction = 5
Number of Element in Y direction = 20
Number of Blocks in Y direction = 20

Node Information

Node#, Coordinates, Parameters, Variables

1	212.85	459.90	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
2	213.00	460.00	0.059417	-0.023791	0.0037216	-0.00068979	-0.00068979	0.0022751	0.0000	0.059998	0.0000
3	215.55	461.75	0.33073	-0.13175	0.019175	-0.0020616	-0.0020616	0.014821	0.0000	0.31000	0.0000
4	218.10	463.50	0.42486	-0.16561	0.024813	-0.0030464	-0.0030464	0.018186	0.0000	0.51000	0.0000
5	220.65	465.25	0.44921	-0.17198	0.025850	-0.0030060	-0.0030060	0.019149	0.0000	0.55000	0.0000
6	223.20	467.00	0.40760	-0.15478	0.023993	-0.0026562	-0.0026562	0.018007	0.0000	0.48000	0.0000
7	225.75	468.75	0.40772	-0.15446	0.023997	-0.0026515	-0.0026515	0.018003	0.0000	0.60000	0.0000
8	228.30	471.08	0.45167	-0.17113	0.026744	-0.0033139	-0.0033139	0.019253	0.0000	0.66000	0.0000
9	230.85	472.25	0.47975	-0.17885	0.027901	-0.0029466	-0.0029466	0.021095	0.0000	0.73000	0.0000
10	233.40	474.00	0.47279	-0.17459	0.027799	-0.0032503	-0.0032503	0.020197	0.0000	0.58000	0.0000
11	235.95	475.75	0.45998	-0.16887	0.026931	-0.0029126	-0.0029126	0.020067	0.0000	0.63000	0.0000
12	238.50	477.50	0.49128	-0.17936	0.028822	-0.0032222	-0.0032222	0.021173	0.0000	0.48000	0.0000
13	241.05	479.25	0.44441	-0.16195	0.025945	-0.0028961	-0.0028961	0.019053	0.0000	0.54000	0.0000
14	243.60	481.00	0.48040	-0.17417	0.027837	-0.0032049	-0.0032049	0.020159	0.0000	0.53000	0.0000
15	246.15	482.75	0.45902	-0.16565	0.026963	-0.0028745	-0.0028745	0.020035	0.0000	0.42000	0.0000
16	248.70	484.50	0.41272	-0.14962	0.024072	-0.0025643	-0.0025643	0.017928	0.0000	0.40000	0.0000
17	251.25	486.25	0.31378	-0.11150	0.018325	-0.0018920	-0.0018920	0.013673	0.0000	0.74000	0.0000

Example Output of a Mesh

18	253.80	488.00	0.17010	-0.052456	0.0097417	-0.00084436	-0.00084436	0.0072642	0.0000	0.15000	0.0000
19	256.35	488.76	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
20	258.90	491.50	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
21	261.45	493.25	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
22	337.93	409.95	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
23	338.30	411.38	0.11519	-0.045900	0.0080883	-0.0010310	-0.0010310	0.0059116	0.0000	0.18000	0.0000
24	339.87	413.62	0.28182	-0.11130	0.017753	-0.0018767	-0.0018767	0.013742	0.0000	0.43500	0.0000
25	341.45	415.87	0.30794	-0.12004	0.019147	-0.0021996	-0.0021996	0.014361	0.0000	0.50500	0.0000
26	343.03	418.13	0.35211	-0.13470	0.022038	-0.0025003	-0.0025003	0.016459	0.0000	0.57500	0.0000
27	344.60	420.38	0.29462	-0.11148	0.018246	-0.0019849	-0.0019849	0.013752	0.0000	0.59500	0.0000
28	346.17	422.63	0.35748	-0.13467	0.023007	-0.0026403	-0.0026403	0.016994	0.0000	0.64999	0.0000
29	347.75	425.17	0.37121	-0.13935	0.023890	-0.0029621	-0.0029621	0.017111	0.0000	0.68000	0.0000
30	349.32	427.13	0.39794	-0.14779	0.025031	-0.0026119	-0.0026119	0.018968	0.0000	0.73500	0.0000
31	350.90	429.38	0.36987	-0.13577	0.023489	-0.0027494	-0.0027494	0.017008	0.0000	0.69000	0.0000
32	352.48	431.63	0.38665	-0.14084	0.024502	-0.0027332	-0.0027332	0.017994	0.0000	0.68500	0.0000
33	354.05	433.87	0.40532	-0.14654	0.025960	-0.0028785	-0.0028785	0.019038	0.0000	0.59000	0.0000
34	355.62	436.12	0.37971	-0.13652	0.024028	-0.0027084	-0.0027084	0.017469	0.0000	0.58000	0.0000
35	357.20	438.37	0.39729	-0.14226	0.024978	-0.0028573	-0.0028573	0.018021	0.0000	0.61500	0.0000
36	358.78	440.62	0.39660	-0.14123	0.025491	-0.0028450	-0.0028450	0.018515	0.0000	0.51500	0.0000
37	360.35	442.87	0.36144	-0.12819	0.023108	-0.0025213	-0.0025213	0.016893	0.0000	0.50000	0.0000
38	361.92	445.12	0.29765	-0.10462	0.019290	-0.0020335	-0.0020335	0.014219	0.0000	0.65750	0.0000
39	363.50	447.37	0.22682	-0.078443	0.014968	-0.0015460	-0.0015460	0.011033	0.0000	0.33000	0.0000
40	365.07	449.13	0.11687	-0.039922	0.0081859	-0.00091716	-0.00091716	0.0058141	0.0000	0.20750	0.0000
41	366.65	451.87	0.087132	-0.029514	0.0062945	-0.00060739	-0.00060739	0.0047071	0.0000	0.099998	0.0000
42	368.23	454.12	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
43	463.00	360.00	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
44	463.60	362.75	0.17756	-0.048264	0.012725	-0.0010119	-0.0010119	0.0092778	0.0000	0.30000	0.0000
45	464.20	365.50	0.24142	-0.064943	0.016661	-0.0012536	-0.0012536	0.012338	0.0000	0.56000	0.0000
46	464.80	368.25	0.19819	-0.052391	0.013739	-0.00098776	-0.00098776	0.010263	0.0000	0.50000	0.0000
47	465.40	371.00	0.26415	-0.068974	0.018614	-0.0014659	-0.0014659	0.013383	0.0000	0.60000	0.0000
48	466.00	373.75	0.18778	-0.048723	0.012748	-0.00097186	-0.00097186	0.0092548	0.0000	0.71000	0.0000
49	466.60	376.50	0.31759	-0.081990	0.022503	-0.0019370	-0.0019370	0.015499	0.0000	0.70000	0.0000
50	467.20	379.25	0.30034	-0.076770	0.021512	-0.0019202	-0.0019202	0.014491	0.0000	0.70000	0.0000
51	467.80	382.00	0.32675	-0.082490	0.022583	-0.0016622	-0.0016622	0.016418	0.0000	0.74000	0.0000
52	468.40	384.75	0.27556	-0.068689	0.019587	-0.0016419	-0.0016419	0.013410	0.0000	0.80000	0.0000
53	469.00	387.50	0.32332	-0.079722	0.022544	-0.0018603	-0.0018603	0.015458	0.0000	0.74000	0.0000
54	469.60	390.25	0.32934	-0.080338	0.023553	-0.0018426	-0.0018426	0.016448	0.0000	0.70000	0.0000
55	470.20	393.00	0.32464	-0.078523	0.022560	-0.0018288	-0.0018288	0.015442	0.0000	0.62000	0.0000
56	470.80	395.75	0.32373	-0.078028	0.022563	-0.0018231	-0.0018231	0.015439	0.0000	0.70000	0.0000
57	471.40	398.50	0.34424	-0.082533	0.024512	-0.0020411	-0.0020411	0.016488	0.0000	0.61000	0.0000
58	472.00	401.25	0.31904	-0.076142	0.022571	-0.0018071	-0.0018071	0.015431	0.0000	0.60000	0.0000
59	472.60	404.00	0.28986	-0.069173	0.020626	-0.0015814	-0.0015814	0.014377	0.0000	0.57500	0.0000

Example Output of a Mesh

60	473.20	406.75	0.29385	-0.069713	0.020631	-0.0015731	-0.0015731	0.014373	0.0000	0.51000	0.0000
61	473.80	409.50	0.24035	-0.056928	0.016680	-0.0013449	-0.0013449	0.011320	0.0000	0.41500	0.0000
62	474.40	412.25	0.17876	-0.043575	0.012776	-0.00091983	-0.00091983	0.0092269	0.0000	0.20000	0.0000
63	475.00	415.00	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
64	588.00	340.00	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
65	588.30	342.87	0.16994	-0.026590	0.011906	-0.00061058	-0.00061058	0.0080988	0.0000	0.27000	0.0000
66	588.60	345.75	0.23879	-0.036083	0.016391	-0.00073827	-0.00073827	0.011617	0.0000	0.40500	0.0000
67	588.90	348.62	0.23200	-0.034190	0.015893	-0.00072079	-0.00072079	0.011108	0.0000	0.43000	0.0000
68	589.20	351.50	0.28644	-0.042010	0.019867	-0.00093327	-0.00093327	0.013641	0.0000	0.51500	0.0000
69	589.50	354.38	0.31064	-0.045842	0.021351	-0.0010104	-0.0010104	0.014653	0.0000	0.59000	0.0000
70	589.80	357.25	0.36657	-0.053859	0.025811	-0.0012948	-0.0012948	0.017189	0.0000	0.56000	0.0000
71	590.10	360.13	0.34347	-0.049394	0.024333	-0.0011986	-0.0011986	0.016171	0.0000	0.59000	0.0000
72	590.40	363.00	0.36342	-0.051188	0.025340	-0.0011749	-0.0011749	0.017164	0.0000	0.61000	0.0000
73	590.70	365.87	0.37048	-0.051204	0.025842	-0.0011527	-0.0011527	0.017661	0.0000	0.65000	0.0000
74	591.00	368.75	0.39389	-0.053466	0.027332	-0.0012660	-0.0012660	0.018177	0.0000	0.59000	0.0000
75	591.30	371.62	0.42472	-0.056763	0.029831	-0.0013133	-0.0013133	0.020180	0.0000	0.58500	0.0000
76	591.60	374.50	0.39457	-0.052152	0.027341	-0.0012339	-0.0012339	0.018168	0.0000	0.54501	0.0000
77	591.90	377.37	0.41985	-0.055478	0.029331	-0.0012990	-0.0012990	0.019672	0.0000	0.60500	0.0000
78	592.20	380.25	0.41886	-0.055374	0.029331	-0.0012996	-0.0012996	0.019673	0.0000	0.55500	0.0000
79	592.50	383.12	0.41734	-0.055397	0.029330	-0.0013047	-0.0013047	0.019674	0.0000	0.55500	0.0000
80	592.80	386.00	0.40436	-0.054383	0.028335	-0.0012548	-0.0012548	0.019174	0.0000	0.54750	0.0000
81	593.10	388.88	0.40967	-0.056192	0.028822	-0.0013466	-0.0013466	0.019190	0.0000	0.50000	0.0000
82	593.40	391.75	0.35207	-0.049281	0.024344	-0.0010977	-0.0010977	0.016656	0.0000	0.43750	0.0000
83	593.70	394.62	0.26732	-0.037961	0.018886	-0.00083579	-0.00083579	0.013119	0.0000	0.25000	0.0000
84	594.00	397.50	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
85	713.00	320.00	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
86	713.00	323.00	0.15946	-0.013136	0.010975	-0.00032711	-0.00032711	0.0070307	0.0000	0.24000	0.0000
87	713.00	326.00	0.23219	-0.019429	0.015964	-0.00041525	-0.00041525	0.011036	0.0000	0.25000	0.0000
88	713.00	329.00	0.26301	-0.022840	0.017953	-0.00051685	-0.00051685	0.012046	0.0000	0.36000	0.0000
89	713.00	332.00	0.30475	-0.027664	0.020946	-0.00063058	-0.00063058	0.014057	0.0000	0.43000	0.0000
90	713.00	335.00	0.43193	-0.042317	0.029904	-0.00097062	-0.00097062	0.020092	0.0000	0.47000	0.0000
91	713.00	338.00	0.41074	-0.043145	0.028890	-0.0010392	-0.0010392	0.019106	0.0000	0.42000	0.0000
92	713.00	341.00	0.38159	-0.042985	0.026888	-0.0010014	-0.0010014	0.018111	0.0000	0.48000	0.0000
93	713.00	344.00	0.39441	-0.045296	0.027870	-0.0011337	-0.0011337	0.018128	0.0000	0.48000	0.0000
94	713.00	347.00	0.46110	-0.051808	0.031873	-0.0011090	-0.0011090	0.022128	0.0000	0.50000	0.0000
95	713.00	350.00	0.45927	-0.050175	0.031868	-0.0011869	-0.0011869	0.021133	0.0000	0.44000	0.0000
96	713.00	353.00	0.51519	-0.053859	0.035873	-0.0012410	-0.0012410	0.024132	0.0000	0.47000	0.0000
97	713.00	356.00	0.45971	-0.045958	0.031889	-0.0010882	-0.0010882	0.021113	0.0000	0.47000	0.0000
98	713.00	359.00	0.51162	-0.049456	0.035892	-0.0011494	-0.0011494	0.024113	0.0000	0.51000	0.0000
99	713.00	362.00	0.48892	-0.045145	0.033904	-0.0010066	-0.0010066	0.023096	0.0000	0.50000	0.0000
100	713.00	365.00	0.51207	-0.044465	0.035913	-0.0010343	-0.0010343	0.024092	0.0000	0.51000	0.0000
101	713.00	368.00	0.51631	-0.041815	0.035925	-0.00096562	-0.00096562	0.024080	0.0000	0.52000	0.0000

Example Output of a Mesh

102	713.00	371.00	0.52346	-0.040251	0.036926	-0.00099380	-0.00099380	0.024078	0.0000	0.49000	0.0000
103	713.00	374.00	0.46264	-0.035495	0.031939	-0.00076233	-0.00076233	0.022062	0.0000	0.46000	0.0000
104	713.00	377.00	0.35491	-0.027912	0.024952	-0.00062555	-0.00062555	0.017047	0.0000	0.30000	0.0000
105	713.00	380.00	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
106	838.00	318.50	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
107	838.00	321.50	0.13350	-0.000841	0.0090026	-1.8891e-05	-1.8891e-05	0.0060045	0.0000	0.16500	0.0000
108	838.00	324.50	0.47644	-0.007756	0.030996	-0.00016266	-0.00016266	0.021006	0.0000	0.29000	0.0000
109	838.00	327.50	0.58375	-0.017088	0.037991	-0.00036551	-0.00036551	0.025516	0.0000	0.38000	0.0000
110	838.00	330.50	0.63510	-0.022560	0.041487	-0.00047908	-0.00047908	0.028017	0.0000	0.46500	0.0000
111	838.00	333.50	0.69435	-0.030092	0.045969	-0.00067029	-0.00067029	0.030531	0.0000	0.50000	0.0000
112	838.00	336.50	0.64347	-0.036446	0.042953	-0.00081842	-0.00081842	0.028550	0.0000	0.44000	0.0000
113	838.00	339.50	0.59152	-0.041971	0.039440	-0.00091783	-0.00091783	0.026570	0.0000	0.43000	0.0000
114	838.00	342.50	0.59445	-0.049460	0.039412	-0.0011159	-0.0011159	0.026094	0.0000	0.38500	0.0000
115	838.00	345.50	0.48935	-0.045938	0.032901	-0.0010230	-0.0010230	0.022099	0.0000	0.36501	0.0000
116	838.00	348.50	0.54561	-0.051202	0.036390	-0.0011625	-0.0011625	0.024111	0.0000	0.30500	0.0000
117	838.00	351.50	0.55957	-0.046495	0.037420	-0.0010319	-0.0010319	0.025087	0.0000	0.34000	0.0000
118	838.00	354.50	0.53900	-0.040295	0.035936	-0.00089220	-0.00089220	0.024069	0.0000	0.36501	0.0000
119	838.00	357.50	0.43836	-0.031623	0.029955	-0.00071782	-0.00071782	0.020056	0.0000	0.35500	0.0000
120	838.00	360.50	0.39818	-0.025601	0.026963	-0.00057595	-0.00057595	0.018042	0.0000	0.31500	0.0000
121	838.00	363.50	0.53190	-0.025318	0.035976	-0.00056995	-0.00056995	0.024029	0.0000	0.31001	0.0000
122	838.00	366.50	0.50623	-0.016519	0.033992	-0.00037478	-0.00037478	0.022519	0.0000	0.35500	0.0000
123	838.00	369.50	0.51591	-0.009980	0.034999	-0.00023208	-0.00023208	0.023007	0.0000	0.39000	0.0000
124	838.00	372.50	0.56045	-0.00768	0.037000	-0.00016443	-0.00016443	0.025004	0.0000	0.38000	0.0000
125	838.00	375.50	0.33447	-0.004462	0.022498	-0.00010004	-0.00010004	0.015001	0.0000	0.29501	0.0000
126	838.00	378.50	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
127	963.00	317.00	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
128	963.00	320.00	0.10510	-0.020091	0.0069330	-0.00037005	-0.00037005	0.0050680	0.0000	0.089996	0.0000
129	963.00	323.00	0.70687	-0.13690	0.045455	-0.0027999	-0.0027999	0.031540	0.0000	0.33000	0.0000
130	963.00	326.00	0.88737	-0.17260	0.057306	-0.0035604	-0.0035604	0.039694	0.0000	0.40000	0.0000
131	963.00	329.00	0.94726	-0.18416	0.061268	-0.0037460	-0.0037460	0.042728	0.0000	0.50000	0.0000
132	963.00	332.00	0.93826	-0.18330	0.061225	-0.0039511	-0.0039511	0.041772	0.0000	0.53000	0.0000
133	963.00	335.00	0.85952	-0.16914	0.056291	-0.0035990	-0.0035990	0.038710	0.0000	0.46000	0.0000
134	963.00	338.00	0.78656	-0.15660	0.051354	-0.0032552	-0.0032552	0.035652	0.0000	0.38000	0.0000
135	963.00	341.00	0.78007	-0.15846	0.050329	-0.0033176	-0.0033176	0.034670	0.0000	0.29000	0.0000
136	963.00	344.00	0.50786	-0.10693	0.033487	-0.0024180	-0.0024180	0.022512	0.0000	0.23000	0.0000
137	963.00	347.00	0.61964	-0.13421	0.040373	-0.0028964	-0.0028964	0.027628	0.0000	0.17000	0.0000
138	963.00	350.00	0.59116	-0.12868	0.038413	-0.0027019	-0.0027019	0.026589	0.0000	0.21000	0.0000
139	963.00	353.00	0.60364	-0.13704	0.039364	-0.0028068	-0.0028068	0.027638	0.0000	0.26000	0.0000
140	963.00	356.00	0.35337	-0.091385	0.023500	-0.0019400	-0.0019400	0.016501	0.0000	0.20000	0.0000
141	963.00	359.00	0.29800	-0.073794	0.019592	-0.0016324	-0.0016324	0.013405	0.0000	0.13000	0.0000
142	963.00	362.00	0.53808	-0.11863	0.035447	-0.0025232	-0.0025232	0.024558	0.0000	0.11000	0.0000
143	963.00	365.00	0.48406	-0.10349	0.031517	-0.0022476	-0.0022476	0.021484	0.0000	0.19000	0.0000

Example Output of a Mesh

144	963.00	368.00	0.49602	-0.10495	0.032526	-0.0022264	-0.0022264	0.022474	0.0000	0.29000	0.0000
145	963.00	371.00	0.64343	-0.13281	0.041428	-0.0027717	-0.0027717	0.028572	0.0000	0.30000	0.0000
146	963.00	374.00	0.30685	-0.061737	0.019724	-0.0013528	-0.0013528	0.013273	0.0000	0.29000	0.0000
147	963.00	377.00	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
148	1088.0	268.50	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
149	1088.5	271.25	0.11676	-0.046019	0.0080950	-0.0010254	-0.0010254	0.0058973	0.0000	0.14999	0.0000
150	1089.0	274.00	0.44088	-0.16996	0.029208	-0.0033570	-0.0033570	0.021794	0.0000	0.34000	0.0000
151	1089.5	276.75	0.52802	-0.19821	0.035018	-0.0039461	-0.0039461	0.025987	0.0000	0.43000	0.0000
152	1090.0	279.50	0.57116	-0.20988	0.037952	-0.0042064	-0.0042064	0.028050	0.0000	0.49500	0.0000
153	1090.5	282.25	0.59538	-0.21496	0.039443	-0.0043134	-0.0043134	0.029054	0.0000	0.54000	0.0000
154	1091.0	285.00	0.57069	-0.20083	0.038072	-0.0040709	-0.0040709	0.027937	0.0000	0.54500	0.0000
155	1091.5	287.75	0.53326	-0.18365	0.035678	-0.0038475	-0.0038475	0.025831	0.0000	0.56000	0.0000
156	1092.0	290.50	0.52592	-0.17884	0.034759	-0.0036580	-0.0036580	0.025246	0.0000	0.54000	0.0000
157	1092.5	293.25	0.40621	-0.13798	0.027017	-0.0028927	-0.0028927	0.019484	0.0000	0.53500	0.0000
158	1093.0	296.00	0.45950	-0.15980	0.030368	-0.0032593	-0.0032593	0.022130	0.0000	0.54500	0.0000
159	1093.5	298.75	0.43892	-0.15806	0.029297	-0.0033465	-0.0033465	0.021209	0.0000	0.62000	0.0000
160	1094.0	301.50	0.43615	-0.16124	0.029300	-0.0032531	-0.0032531	0.021703	0.0000	0.63000	0.0000
161	1094.5	304.25	0.31585	-0.11890	0.021134	-0.0023093	-0.0023093	0.015869	0.0000	0.60000	0.0000
162	1095.0	307.00	0.25791	-0.099681	0.017218	-0.0020165	-0.0020165	0.012780	0.0000	0.56500	0.0000
163	1095.5	309.75	0.37344	-0.15282	0.025213	-0.0031556	-0.0031556	0.018793	0.0000	0.47000	0.0000
164	1096.0	312.50	0.33599	-0.14001	0.022316	-0.0028389	-0.0028389	0.016686	0.0000	0.41500	0.0000
165	1096.5	315.25	0.28648	-0.11976	0.019459	-0.0024896	-0.0024896	0.014545	0.0000	0.44500	0.0000
166	1097.0	318.00	0.32130	-0.13369	0.021392	-0.0026600	-0.0026600	0.016106	0.0000	0.35001	0.0000
167	1097.5	320.75	0.14452	-0.060056	0.0094799	-0.0012382	-0.0012382	0.0070148	0.0000	0.14500	0.0000
168	1098.0	323.50	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
169	1213.0	220.00	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
170	1214.0	222.50	0.12297	-0.074927	0.0089177	-0.0017798	-0.0017798	0.0070812	0.0000	0.21000	0.0000
171	1215.0	225.00	0.19253	-0.11643	0.013661	-0.0022128	-0.0022128	0.011340	0.0000	0.35000	0.0000
172	1216.0	227.50	0.19239	-0.11473	0.013688	-0.0021983	-0.0021983	0.011313	0.0000	0.46000	0.0000
173	1217.0	230.00	0.21732	-0.12758	0.015461	-0.0026181	-0.0026181	0.012539	0.0000	0.49000	0.0000
174	1218.0	232.50	0.26856	-0.15484	0.018501	-0.0025949	-0.0025949	0.015496	0.0000	0.55000	0.0000
175	1219.0	235.00	0.29010	-0.16552	0.020283	-0.0030147	-0.0030147	0.016719	0.0000	0.63000	0.0000
176	1220.0	237.50	0.28340	-0.16111	0.020048	-0.0034385	-0.0034385	0.015955	0.0000	0.74000	0.0000
177	1221.0	240.00	0.27398	-0.15544	0.019298	-0.0030060	-0.0030060	0.015705	0.0000	0.79000	0.0000
178	1222.0	242.50	0.29484	-0.16729	0.020298	-0.0030061	-0.0030061	0.016705	0.0000	0.84000	0.0000
179	1223.0	245.00	0.29452	-0.16786	0.020286	-0.0030131	-0.0030131	0.016717	0.0000	0.92000	0.0000
180	1224.0	247.50	0.28461	-0.16304	0.020026	-0.0034519	-0.0034519	0.015977	0.0000	1.0300	0.0000
181	1225.0	250.00	0.26958	-0.15507	0.019262	-0.0030271	-0.0030271	0.015741	0.0000	1.0000	0.0000
182	1226.0	252.50	0.2684	-0.15419	0.018496	-0.0025977	-0.0025977	0.015501	0.0000	1.0000	0.0000
183	1227.0	255.00	0.21242	-0.12208	0.014728	-0.0021764	-0.0021764	0.012273	0.0000	1.0000	0.0000
184	1228.0	257.50	0.22057	-0.12693	0.015453	-0.0026225	-0.0026225	0.012546	0.0000	0.83000	0.0000
185	1229.0	260.00	0.20067	-0.11841	0.013709	-0.0021871	-0.0021871	0.011293	0.0000	0.64000	0.0000

Example Output of a Mesh

186	1230.0	262.50	0.098543	-0.057314	0.0072434	-0.0013064	-0.0013064	0.0057571	0.0000	0.60000	0.0000
187	1231.0	265.00	0.034803	-0.017604	0.0027947	-0.00040254	-0.00040254	0.0022025	0.0000	0.40000	0.0000
188	1232.0	267.50	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
189	1233.0	270.00	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
190	1338.0	110.00	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
191	1339.5	112.50	0.10247	-0.090175	0.0092568	-0.0019827	-0.0019827	0.0087486	0.0000	0.25500	0.0000
192	1340.9	115.00	0.13921	-0.12184	0.012051	-0.0022271	-0.0022271	0.011455	0.0000	0.38000	0.0000
193	1342.3	117.50	0.10241	-0.087970	0.0082255	-0.0014820	-0.0014820	0.0077734	0.0000	0.55000	0.0000
194	1343.8	120.00	0.10834	-0.091026	0.0085530	-0.0017247	-0.0017247	0.0079494	0.0000	0.58500	0.0000
195	1345.3	122.50	0.13398	-0.11103	0.010373	-0.0019641	-0.0019641	0.0096309	0.0000	0.62500	0.0000
196	1346.7	125.00	0.14314	-0.11878	0.011165	-0.0022120	-0.0022120	0.010335	0.0000	0.77500	0.0000
197	1348.2	127.50	0.22868	-0.19030	0.020232	-0.0039350	-0.0039350	0.018778	0.0000	0.83000	0.0000
198	1349.6	130.00	0.24051	-0.19953	0.021537	-0.0041765	-0.0041765	0.019967	0.0000	0.76500	0.0000
199	1351.0	132.50	0.25953	-0.21419	0.023061	-0.0041724	-0.0041724	0.021449	0.0000	0.70500	0.0000
200	1352.5	135.00	0.28221	-0.23148	0.025183	-0.0046613	-0.0046613	0.023324	0.0000	0.72000	0.0000
201	1354.0	137.50	0.26667	-0.21731	0.023713	-0.0046515	-0.0046515	0.021796	0.0000	0.79000	0.0000
202	1355.4	140.00	0.28090	-0.22756	0.025540	-0.0048922	-0.0048922	0.023464	0.0000	0.78500	0.0000
203	1356.8	142.50	0.28516	-0.22943	0.025770	-0.0046392	-0.0046392	0.023737	0.0000	0.79000	0.0000
204	1358.3	145.00	0.25131	-0.20052	0.023002	-0.0043890	-0.0043890	0.021004	0.0000	0.79000	0.0000
205	1359.8	147.50	0.24111	-0.19085	0.021728	-0.0041360	-0.0041360	0.019776	0.0000	0.72000	0.0000
206	1361.2	150.00	0.23648	-0.18541	0.021454	-0.0038823	-0.0038823	0.019546	0.0000	0.61500	0.0000
207	1362.7	152.50	0.16102	-0.12526	0.015238	-0.0029099	-0.0029099	0.013761	0.0000	0.60000	0.0000
208	1364.1	155.00	0.15105	-0.11772	0.015236	-0.0029075	-0.0029075	0.013771	0.0000	0.49001	0.0000
209	1365.5	157.50	0.11127	-0.088215	0.011263	-0.0021913	-0.0021913	0.010236	0.0000	0.28000	0.0000
210	1367.0	160.00	0.091371	-0.073876	0.0094213	-0.0019546	-0.0019546	0.0085841	0.0000	0.28500	0.0000
211	1463.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
212	1464.9	2.5000	0.096943	-0.085103	0.010261	-0.0019821	-0.0019821	0.0097432	0.0000	0.30000	0.0000
213	1466.8	5.0000	0.10912	-0.095480	0.011267	-0.0019812	-0.0019812	0.010736	0.0000	0.41000	0.0000
214	1468.7	7.5000	0.034810	-0.030067	0.0035707	-0.00049441	-0.00049441	0.0034254	0.0000	0.64000	0.0000
215	1470.6	10.000	0.023712	-0.019966	0.0025837	-0.00049243	-0.00049243	0.0024135	0.0000	0.68000	0.0000
216	1472.5	12.500	0.029907	-0.023447	0.0032369	-0.00097057	-0.00097057	0.0027598	0.0000	0.70000	0.0000
217	1474.4	15.000	0.030590	-0.022548	0.0032941	-0.00095470	-0.00095470	0.0027026	0.0000	0.92000	0.0000
218	1476.3	17.500	0.20868	-0.16974	0.021815	-0.0039178	-0.0039178	0.020185	0.0000	0.92000	0.0000
219	1478.2	20.000	0.23912	-0.19728	0.024949	-0.0049100	-0.0049100	0.023048	0.0000	0.74000	0.0000
220	1480.1	22.500	0.25829	-0.21175	0.026981	-0.0049002	-0.0049002	0.025021	0.0000	0.57000	0.0000
221	1482.0	25.000	0.30337	-0.24667	0.031227	-0.0058743	-0.0058743	0.028778	0.0000	0.52000	0.0000
222	1483.9	27.500	0.28032	-0.22587	0.028669	-0.0053712	-0.0053712	0.026331	0.0000	0.55000	0.0000
223	1485.8	30.000	0.32181	-0.25726	0.032933	-0.0063407	-0.0063407	0.030070	0.0000	0.57000	0.0000
224	1487.7	32.500	0.33016	-0.26282	0.033959	-0.0063347	-0.0063347	0.031044	0.0000	0.58000	0.0000
225	1489.6	35.000	0.31157	-0.24604	0.032009	-0.0063231	-0.0063231	0.028995	0.0000	0.58000	0.0000
226	1491.5	37.500	0.28274	-0.22122	0.028823	-0.0053355	-0.0053355	0.026178	0.0000	0.61000	0.0000
227	1493.4	40.000	0.29034	-0.22611	0.029846	-0.0053295	-0.0053295	0.027153	0.0000	0.59000	0.0000

Example Output of a Mesh

228	1495.3	42.500	0.23240	-0.18007	0.023623	-0.0043587	-0.0043587	0.021375	0.0000	0.60000	0.0000
229	1497.2	45.000	0.27148	-0.21127	0.027851	-0.0053285	-0.0053285	0.025150	0.0000	0.58000	0.0000
230	1499.1	47.500	0.22312	-0.17571	0.022555	-0.0043759	-0.0043759	0.020444	0.0000	0.56000	0.0000
231	1501.0	50.000	0.18164	-0.14911	0.018781	-0.0039249	-0.0039249	0.017222	0.0000	0.57000	0.0000

Element Information

Element #, vtype, gtype, nodes

1	211	211	22	23	2	1	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
2	211	211	23	24	3	2	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
3	211	211	24	25	4	3	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
4	211	211	25	26	5	4	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
5	211	211	26	27	6	5	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
6	211	211	27	28	7	6	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
7	211	211	28	29	8	7	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
8	211	211	29	30	9	8	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
9	211	211	30	31	10	9	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
10	211	211	31	32	11	10	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
11	211	211	32	33	12	11	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
12	211	211	33	34	13	12	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
13	211	211	34	35	14	13	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
14	211	211	35	36	15	14	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
15	211	211	36	37	16	15	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
16	211	211	37	38	17	16	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
17	211	211	38	39	18	17	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
18	211	211	39	40	19	18	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
19	211	211	40	41	20	19	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
20	211	211	41	42	21	20	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
21	211	211	43	44	23	22	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
22	211	211	44	45	24	23	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
23	211	211	45	46	25	24	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
24	211	211	46	47	26	25	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
25	211	211	47	48	27	26	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
26	211	211	48	49	28	27	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
27	211	211	49	50	29	28	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
28	211	211	50	51	30	29	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
29	211	211	51	52	31	30	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
30	211	211	52	53	32	31	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
31	211	211	53	54	33	32	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
32	211	211	54	55	34	33	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
33	211	211	55	56	35	34	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Example Output of a Mesh

34	211	211	56	57	36	35	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
35	211	211	57	58	37	36	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
36	211	211	58	59	38	37	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
37	211	211	59	60	39	38	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
38	211	211	60	61	40	39	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
39	211	211	61	62	41	40	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
40	211	211	62	63	42	41	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
41	211	211	64	65	44	43	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
42	211	211	65	66	45	44	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
43	211	211	66	67	46	45	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
44	211	211	67	68	47	46	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
45	211	211	68	69	48	47	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
46	211	211	69	70	49	48	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
47	211	211	70	71	50	49	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
48	211	211	71	72	51	50	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
49	211	211	72	73	52	51	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
50	211	211	73	74	53	52	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
51	211	211	74	75	54	53	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
52	211	211	75	76	55	54	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
53	211	211	76	77	56	55	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
54	211	211	77	78	57	56	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
55	211	211	78	79	58	57	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
56	211	211	79	80	59	58	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
57	211	211	80	81	60	59	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
58	211	211	81	82	61	60	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
59	211	211	82	83	62	61	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
60	211	211	83	84	63	62	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
61	211	211	85	86	65	64	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
62	211	211	86	87	66	65	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
63	211	211	87	88	67	66	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
64	211	211	88	89	68	67	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
65	211	211	89	90	69	68	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
66	211	211	90	91	70	69	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
67	211	211	91	92	71	70	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
68	211	211	92	93	72	71	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
69	211	211	93	94	73	72	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
70	211	211	94	95	74	73	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
71	211	211	95	96	75	74	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
72	211	211	96	97	76	75	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
73	211	211	97	98	77	76	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
74	211	211	98	99	78	77	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
75	211	211	99	100	79	78	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Example Output of a Mesh

76	211	211	100	101	80	79	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
77	211	211	101	102	81	80	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
78	211	211	102	103	82	81	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
79	211	211	103	104	83	82	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
80	211	211	104	105	84	83	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
81	211	211	106	107	86	85	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
82	211	211	107	108	87	86	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
83	211	211	108	109	88	87	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
84	211	211	109	110	89	88	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
85	211	211	110	111	90	89	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
86	211	211	111	112	91	90	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
87	211	211	112	113	92	91	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
88	211	211	113	114	93	92	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
89	211	211	114	115	94	93	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
90	211	211	115	116	95	94	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
91	211	211	116	117	96	95	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
92	211	211	117	118	97	96	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
93	211	211	118	119	98	97	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
94	211	211	119	120	99	98	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
95	211	211	120	121	100	99	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
96	211	211	121	122	101	100	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
97	211	211	122	123	102	101	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
98	211	211	123	124	103	102	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
99	211	211	124	125	104	103	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
100	211	211	125	126	105	104	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
101	211	211	127	128	107	106	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
102	211	211	128	129	108	107	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
103	211	211	129	130	109	108	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
104	211	211	130	131	110	109	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
105	211	211	131	132	111	110	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
106	211	211	132	133	112	111	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
107	211	211	133	134	113	112	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
108	211	211	134	135	114	113	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
109	211	211	135	136	115	114	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
110	211	211	136	137	116	115	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
111	211	211	137	138	117	116	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
112	211	211	138	139	118	117	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
113	211	211	139	140	119	118	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
114	211	211	140	141	120	119	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
115	211	211	141	142	121	120	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
116	211	211	142	143	122	121	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
117	211	211	143	144	123	122	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Example Output of a Mesh

118	211	211	144	145	124	123	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
119	211	211	145	146	125	124	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
120	211	211	146	147	126	125	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
121	211	211	148	149	128	127	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
122	211	211	149	150	129	128	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
123	211	211	150	151	130	129	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
124	211	211	151	152	131	130	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
125	211	211	152	153	132	131	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
126	211	211	153	154	133	132	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
127	211	211	154	155	134	133	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
128	211	211	155	156	135	134	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
129	211	211	156	157	136	135	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
130	211	211	157	158	137	136	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
131	211	211	158	159	138	137	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
132	211	211	159	160	139	138	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
133	211	211	160	161	140	139	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
134	211	211	161	162	141	140	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
135	211	211	162	163	142	141	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
136	211	211	163	164	143	142	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
137	211	211	164	165	144	143	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
138	211	211	165	166	145	144	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
139	211	211	166	167	146	145	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
140	211	211	167	168	147	146	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
141	211	211	169	170	149	148	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
142	211	211	170	171	150	149	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
143	211	211	171	172	151	150	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
144	211	211	172	173	152	151	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
145	211	211	173	174	153	152	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
146	211	211	174	175	154	153	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
147	211	211	175	176	155	154	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
148	211	211	176	177	156	155	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
149	211	211	177	178	157	156	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
150	211	211	178	179	158	157	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
151	211	211	179	180	159	158	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
152	211	211	180	181	160	159	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
153	211	211	181	182	161	160	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
154	211	211	182	183	162	161	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
155	211	211	183	184	163	162	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
156	211	211	184	185	164	163	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
157	211	211	185	186	165	164	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
158	211	211	186	187	166	165	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
159	211	211	187	188	167	166	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Example Output of a Mesh

160	211	211	188	189	168	167	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
161	211	211	190	191	170	169	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
162	211	211	191	192	171	170	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
163	211	211	192	193	172	171	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
164	211	211	193	194	173	172	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
165	211	211	194	195	174	173	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
166	211	211	195	196	175	174	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
167	211	211	196	197	176	175	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
168	211	211	197	198	177	176	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
169	211	211	198	199	178	177	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
170	211	211	199	200	179	178	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
171	211	211	200	201	180	179	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
172	211	211	201	202	181	180	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
173	211	211	202	203	182	181	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
174	211	211	203	204	183	182	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
175	211	211	204	205	184	183	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
176	211	211	205	206	185	184	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
177	211	211	206	207	186	185	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
178	211	211	207	208	187	186	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
179	211	211	208	209	188	187	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
180	211	211	209	210	189	188	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
181	211	211	211	212	191	190	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
182	211	211	212	213	192	191	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
183	211	211	213	214	193	192	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
184	211	211	214	215	194	193	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
185	211	211	215	216	195	194	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
186	211	211	216	217	196	195	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
187	211	211	217	218	197	196	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
188	211	211	218	219	198	197	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
189	211	211	219	220	199	198	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
190	211	211	220	221	200	199	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
191	211	211	221	222	201	200	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
192	211	211	222	223	202	201	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
193	211	211	223	224	203	202	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
194	211	211	224	225	204	203	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
195	211	211	225	226	205	204	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
196	211	211	226	227	206	205	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
197	211	211	227	228	207	206	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
198	211	211	228	229	208	207	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
199	211	211	229	230	209	208	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
200	211	211	230	231	210	209	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Example Output of a Mesh

Boundary Element #, vtype, gtype, nodes, boundary condition codes

1	111	111	1	22	0.0000	0.0000	2
2	111	111	22	43	0.0000	0.0000	2
3	111	111	43	64	0.0000	0.0000	2
4	111	111	64	85	0.0000	0.0000	2
5	111	111	85	106	0.0000	0.0000	2
6	111	111	106	127	0.0000	0.0000	2
7	111	111	127	148	0.0000	0.0000	2
8	111	111	148	169	0.0000	0.0000	2
9	111	111	169	190	0.0000	0.0000	2
10	111	111	190	211	0.0000	0.0000	2
11	111	111	211	212	0.0000	0.0000	2
12	111	111	212	213	0.0000	0.0000	2
13	111	111	213	214	0.0000	0.0000	2
14	111	111	214	215	0.0000	0.0000	2
15	111	111	215	216	0.0000	0.0000	2
16	111	111	216	217	0.0000	0.0000	2
17	111	111	217	218	0.0000	0.0000	2
18	111	111	218	219	0.0000	0.0000	2
19	111	111	219	220	0.0000	0.0000	2
20	111	111	220	221	0.0000	0.0000	2
21	111	111	221	222	0.0000	0.0000	2
22	111	111	222	223	0.0000	0.0000	2
23	111	111	223	224	0.0000	0.0000	2
24	111	111	224	225	0.0000	0.0000	2
25	111	111	225	226	0.0000	0.0000	2
26	111	111	226	227	0.0000	0.0000	2
27	111	111	227	228	0.0000	0.0000	2
28	111	111	228	229	0.0000	0.0000	2
29	111	111	229	230	0.0000	0.0000	2
30	111	111	230	231	0.0000	0.0000	2
31	111	111	231	210	0.0000	0.0000	2
32	111	111	210	189	0.0000	0.0000	2
33	111	111	189	168	0.0000	0.0000	2
34	111	111	168	147	0.0000	0.0000	2
35	111	111	147	126	0.0000	0.0000	2
36	111	111	126	105	0.0000	0.0000	2
37	111	111	105	84	0.0000	0.0000	2
38	111	111	84	63	0.0000	0.0000	2
39	111	111	63	42	0.0000	0.0000	2
40	111	111	42	21	0.0000	0.0000	2

Example Output of a Mesh

41	111	111	21	20	0.0000	0.0000	3
42	111	111	20	19	0.0000	0.0000	3
43	111	111	19	18	0.0000	0.0000	3
44	111	111	18	17	0.0000	0.0000	3
45	111	111	17	16	0.0000	0.0000	3
46	111	111	16	15	0.0000	0.0000	3
47	111	111	15	14	0.0000	0.0000	3
48	111	111	14	13	0.0000	0.0000	3
49	111	111	13	12	0.0000	0.0000	3
50	111	111	12	11	0.0000	0.0000	3
51	111	111	11	10	0.0000	0.0000	3
52	111	111	10	9	0.0000	0.0000	3
53	111	111	9	8	0.0000	0.0000	3
54	111	111	8	7	0.0000	0.0000	3
55	111	111	7	6	0.0000	0.0000	3
56	111	111	6	5	0.0000	0.0000	3
57	111	111	5	4	0.0000	0.0000	3
58	111	111	4	3	0.0000	0.0000	3
59	111	111	3	2	0.0000	0.0000	3
60	111	111	2	1	0.0000	0.0000	3